305-CD-045-001

EOSDIS Core System Project

Flight Operations Segment (FOS) Command Design Specification for the ECS Project

October 1995

Hughes Information Technology Corporation
Upper Marlboro, MD

Flight Operations Segment (FOS) Command Design Specification for the ECS Project

October 1995

Prepared Under Contract NAS5-60000
CDRL Item #046

APPROVED BY

Cal Moore /s/ 9/22/95
Cal Moore, FOS CCB Chairman Date
EOSDIS Core System Project

Hughes Information Technology Corporation
Upper Marlboro, Maryland

305-CD-045-001

This page intentionally left blank.

Preface

This document, one of nineteen, comprises the detailed design specification of the FOS subsystems for Releases A and B of the ECS project. This includes the FOS design to support the AM-1 launch.

The FOS subsystem design specification documents for Releases A and B of the ECS project include:

305-CD-040 FOS Design Specification (Segment Level Design)
305-CD-041 Planning and Scheduling Design Specification
305-CD-042 Command Management Design Specification
305-CD-043 Resource Management Design Specification
305-CD-044 Telemetry Design Specification
305-CD-045 Command Design Specification
305-CD-046 Real-Time Contact Management Design Specification
305-CD-047 Analysis Design Specification
305-CD-048 User Interface Design Specification
305-CD-049 Data Management Design Specification
305-CD-050 Planning and Scheduling Program Design Language (PDL)
305-CD-051 Command Management PDL
305-CD-052 Resource Management PDL
305-CD-053 Telemetry PDL
305-CD-054 Real-Time Contact Management PDL
305-CD-055 Analysis PDL
305-CD-056 User Interface PDL
305-CD-057 Data Management PDL
305-CD-058 Command PDL

Object models presented in this document have been exported directly from CASE tools and in some cases contain too much detail to be easily readable within hard copy page constraints. The reader is encouraged to view these drawings on line using the Portable Document Format (PDF) electronic copy available via the ECS Data Handling System (EDHS) at
URL http://edhs1.gsfc.nasa.gov.

This document is a contract deliverable with an approval code 2. As such, it does not require formal Government approval, however, the Government reserves the right to request changes within 45 days of the initial submittal. Once approved, contractor changes to this document are handled in accordance with Class I and Class II change control requirements described in the EOS Configuration Management Plan, and changes to this document shall be made by document change notice (DCN) or by complete revision.

Any questions should be addressed to:

Data Management Office
The ECS Project Office
Hughes Information Technology Corporation
1616 McCormick Drive
Upper Marlboro, Maryland 20774-5372

Abstract

The FOS Design Specification consists of a set of 19 documents that define the FOS detailed design. The first document, the FOS Segment Level Design, provides an overview of the FOS segment design, the architecture, and analyses and trades. The next nine documents provide the detailed design for each of the nine FOS subsystems. The last nine documents provide the PDL for the nine FOS subsystems. It also allocates the level 4 FOS requirements to the subsystem design.

Keywords: FOS, design, specification, analysis, IST, EOC

This page intentionally left blank.

Change Information Page

List of Effective Pages	
Page Number	**Issue**
Title	Original
iii through xii	Original
1-1 and 1-2	Original
2-1 through 2-4	Original
3-1 through 3-198	Original
AB-1 through AB-8	Original
GL-1 through GL-8	Original

Document History			
Document Number	**Status/Issue**	**Publication Date**	**CCR Number**
305-CD-045-001	Original	October 1995	95-0653

This page intentionally left blank.

Contents

Figures

Tables

Abbreviations and Acronyms

Glossary

This page intentionally left blank.

1. Introduction

1.1 Identification

The contents of this document defines the design specification for the Flight Operations Segment (FOS). Thus, this document addresses the Data Item Description (DID) for CDRL Item 046 305/DV2 under Contract NAS5-60000.

1.2 Scope

The Flight Operations Segment (FOS) Design Specification defines the detailed design of the FOS. It allocates the Level 4 FOS requirements to the subsystem design. It also defines the FOS architectural design. In particular, this document addresses the Data Item Description (DID) for CDRL # 046, the Segment Design Specification.

This document reflects the August 23, 1995 Technical Baseline maintained by the contractor configuration control board in accordance with ECS Technical Direction No. 11, dated December 6, 1994.

1.3 Purpose

The FOS Design Specification consists of a set of 19 documents that define the FOS detailed design. The first document, the FOS Segment Level Design, provides an overview of the FOS segment design, the architecture, and analyses and trades. The next nine documents provide the detailed design for each of the nine FOS subsystems. The last nine documents provide the PDL for the nine FOS subsystems.

1.4 Status and Schedule

This submittal of DID 305/DV2 incorporates the FOS detailed design performed during the Critical Design Review (CDR) time frame. This document is under the ECS Project configuration control.

1.5 Document Organization

305-CD-040 contains the overview, the FOS segment models, the FOS architecture, and FOS analyses and trades performed during the design phase.

305-CD-041 contains the detailed design for Planning and Scheduling Design Specification.

305-CD-042 contains the detailed design for Command Management Design Specification.

305-CD-043 contains the detailed design for Resource Management Design Specification.

305-CD-044 contains the detailed design for Telemetry Design Specification.

305-CD-045 contains the detailed design for Command Design Specification.

305-CD-046 contains the detailed design for Real-Time Contact Management Design Specification.

305-CD-047 contains the detailed design for Analysis Design Specification.

305-CD-048 contains the detailed design for User Interface Design Specification.

305-CD-049 contains the detailed design for Data Management Design Specification.

305-CD-050 contains Planning and Scheduling PDL.

305-CD-051 contains Command Management PDL.

305-CD-052 contains Resource Management PDL.

305-CD-053 contains the Telemetry PDL.

305-CD-054 contains the Real-Time Contact Management PDL.

305-CD-055 contains the Analysis PDL.

305-CD-056 contains the User Interface PDL.

305-CD-057 contains the Data Management PDL.

305-CD-058 contains the Command PDL.

Appendix A of the first document contains the traceability between Level 4 Requirements and the design. The traceability maps the Level 4 requirements to the objects included in the subsystem object models.

Glossary contains the key terms that are included within this design specification.

Abbreviations and acronyms contains an alphabetized list of the definitions for abbreviations and acronyms used within this design specification.

2. Related Documentation

2.1 Parent Document

The parent documents are the documents from which this FOS Design Specification's scope and content are derived.

194-207-SE1-001	System Design Specification for the ECS Project
304-CD-001-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 1: General Requirements
304-CD-004-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 2: Mission Specific

2.2 Applicable Documents

The following documents are referenced within this FOS Design Specification or are directly applicable, or contain policies or other directive matters that are binding upon the content of this volume.

194-219-SE1-020	Interface Requirements Document Between EOSDIS Core System (ECS) and NASA Institutional Support Systems
209-CD-002-002	Interface Control Document Between EOSDIS Core System (ECS) and ASTER Ground Data System, Preliminary
209-CD-003-002	Interface Control Document Between EOSDIS Core System (ECS) and the EOS-AM Project for AM-1 Spacecraft Analysis Software, Preliminary
209-CD-004-002	Data Format Control Document for the Earth Observing System (EOS) AM-1 Project Data Base, Preliminary
209-CD-025-001	ICD Between ECS and AM1 Project Spacecraft Software Development and Validation Facilities (SDVF)
311-CD-001-003	Flight Operations Segment (FOS) Database Design and Database Schema for the ECS Project
502-ICD-JPL/GSFC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Jet Propulsion Laboratory and the Goddard Space Flight Center for GSFC Missions Using the Deep Space Network
530-ICD-NCCDS/MOC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Goddard Space Flight Center Mission Operations Centers and the Network Control Center Data System

530-ICD-NCCDS/POCC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Goddard Space Flight Center Payload Operations Control Centers and the Network Control Center Data System
530-DFCD-NCCDS/POCC	Goddard Space Flight Center/MO&DSD, Data Format control Document Between the Goddard Space Flight Center Payload Operations Control Centers and the Network Control Center Data System
540-041	Interface Control Document (ICD) Between the Earth Observing System (EOS) Communications (Ecom) and the EOS Operations Center (EOC), Review
560-EDOS-0230.0001	Goddard Space Flight Center/MO&DSD, Earth Observing System (EOS) Data and Operations System (EDOS) Data Format Requirements Document (DFRD)
ICD-106	Martin Marietta Corporation, Interface Control Document (ICD) Data Format Control Book for EOS-AM Spacecraft
none	Goddard Space Flight Center, Earth Observing System (EOS) AM-1 Flight Dynamics Facility (FDF) / EOS Operations Center (EOC) Interface Control Document

2.3 Information Documents

2.3.1 Information Document Referenced

The following documents are referenced herein and, amplify or clarify the information presented in this document. These documents are not binding on the content of this FOS Design Specification.

194-201-SE1-001	Systems Engineering Plan for the ECS Project
194-202-SE1-001	Standards and Procedures for the ECS Project
193-208-SE1-001	Methodology for Definition of External Interfaces for the ECS Project
308-CD-001-004	Software Development Plan for the ECS Project
194-501-PA1-001	Performance Assurance Implementation Plan for the ECS Project
194-502-PA1-001	Contractor's Practices & Procedures Referenced in the PAIP for the ECS Project
604-CD-001-004	Operations Concept for the ECS Project: Part 1-- ECS Overview, 6/95
604-CD-002-001	Operations Concept for the ECS project: Part 2B -- ECS Release B, Annotated Outline, 3/95
604-CD-003-001	ECS Operations Concept for the ECS Project: Part 2A -- ECS Release A, Final, 7/95
194-WP-912-001	EOC/ICC Trade Study Report for the ECS Project, Working Paper
194-WP-913-003	User Environment Definition for the ECS Project, Working Paper

194-WP-920-001	An Evaluation of OASIS-CC for Use in the FOS, Working Paper
194-TP-285-001	ECS Glossary of Terms
222-TP-003-006	Release Plan Content Description
none	Hughes Information Technology Company, Technical Proposal for the EOSDIS Core System (ECS), Best and Final Offer
560-EDOS-0211.0001	Goddard Space Flight Center, Interface Requirements Document (IRD) Between the Earth Observing System (EOS) Data and Operations System (EDOS), and the EOS Ground System (EGS) Elements, Preliminary
NHB 2410.9A	NASA Hand Book: Security, Logistics and Industry Relations Division, NASA Security Office: Automated Information Security Handbook

This page intentionally left blank.

3. Command Subsystem

The Command Subsystem provides the capability to: build, validate, uplink and verify real-time commands for the EOS spacecraft and instruments; uplink and verify memory loads for the EOS spacecraft and instruments; and verify execution of stored commands for the EOS spacecraft and instruments during a real-time contact.

3.1 Command Context Description

The context diagram in Figure 3.1-1 depicts the data flows between the FOS Command Subsystem and the internal EOC and external ground system components. Descriptions of data flows are summarized for each components:

Parameter Server: Decommutated spacecraft and instrument telemetry samples are made available to the Command Subsystem process in response to queries. The Command Subsystem uses these samples to verify real-time, load and stored commands.

FOS Telemetry Subsystem: Received Dump notification from Command Subsystem.

FOS Resource Management Subsystem: RMS starts the command tasks running as part of a logical string and then supplies EOC spacecraft contact and commanding session configuration information.

FOS Data Management Subsystem: Command database and stored memory load information may be retrieved from the Data Management Subsystem. Information pertaining to user authorization, command bit pattern definition, validation, and verification is stored in the command database. Spacecraft and instrument commands to be executed autonomously, as well as flight software, may be contained within the stored memory loads. The Data Management Subsystem receives, stores, and forwards to the appropriate subsystems and instrument teams, the command event messages (command uplink status, command verification, command notification and command load status) generated by the Command Subsystem process. Configuration files, part of the database, are read from the DMS. Snapshot files are written to and read from the DMS.

FOS User Interface Subsystem: The User Interface delivers command and load directives to the Command Subsystem. The source of individual directives may be EOC automated ground scripts or FOT input that have been parsed by the User Interface prior to delivery to the Command Subsystem. For each command or load directive, the Command Subsystem performs an authorization check and validation. For transmission of critical spacecraft and instrument commands and loads, the Commanding Subsystem prompts FOT for confirmation via the User Interface, and User Interface delivers FOT's response to the Command Subsystem. Similarly, a prerequisite override prompt gives the operator the option to uplink commands that fail prerequisite state checking. The User Interface also receives and displays command status generated by the Command Subsystem.

EDOS : The Command Subsystem meters out CLTUs to EDOS for uplink to the spacecraft. The Subsystem also receives CLCWs from EDOS.

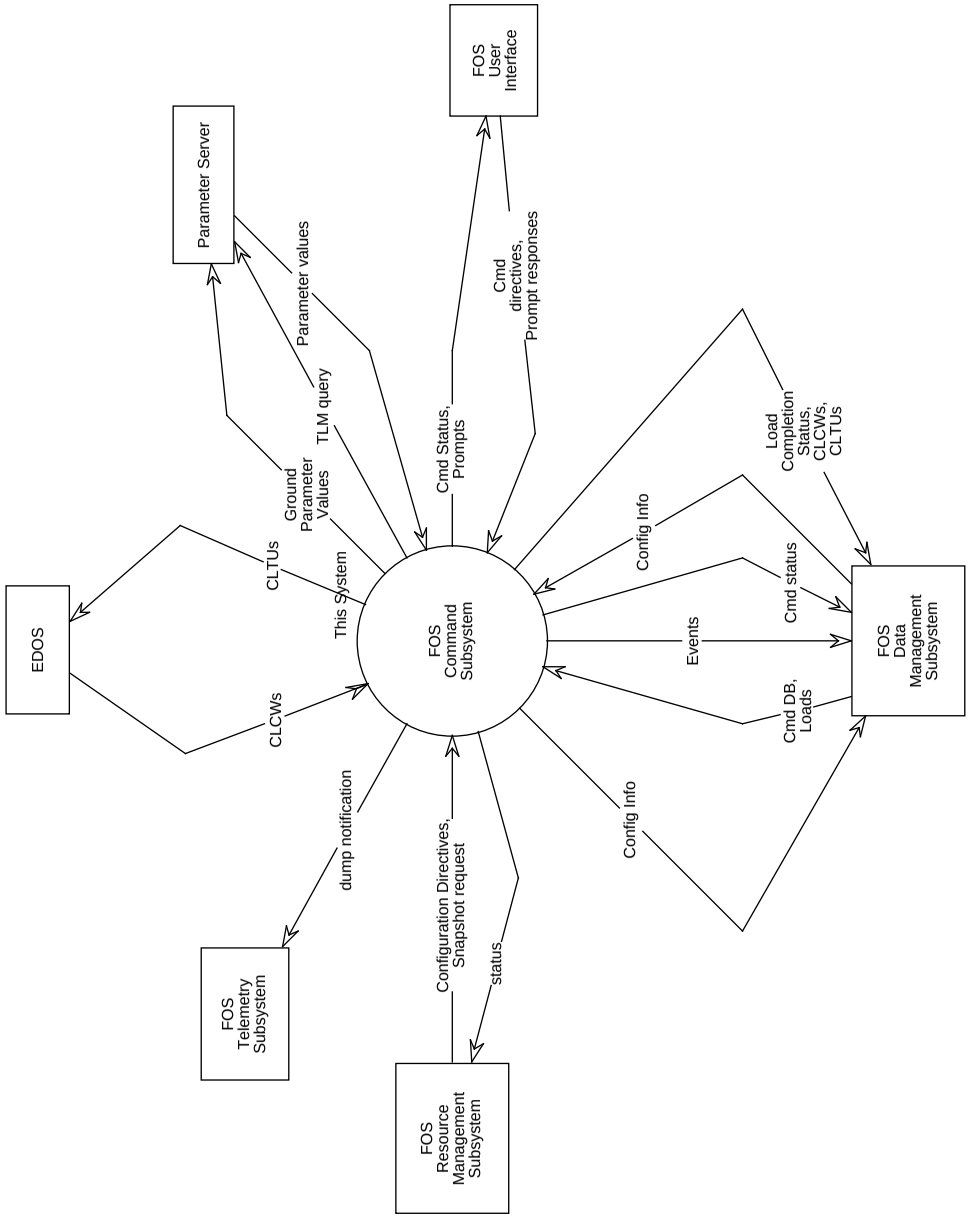

Figure 3.1-1. Command Subsystem Context Diagram

3.2 FormatCommand Description

The FormatCommand process is responsible for the processing of spacecraft commands from the user interface, and formatting them into the format recognized onboard the spacecraft: 1553b format for the AM1 mission. It is also responsible for handling directives from FUI for configuration changes, and execution verification of commands.

3.2.1 FormatCommand Context Description

The context diagram in Figure 3.2.1-1 depicts the data flows between the FOS Command Subsystem and the internal EOC and external ground system components. Descriptions of data flows are summarized for each component:

Parameter Server: Decommutated spacecraft and instrument telemetry samples are made available to the FormatCommand process in response to queries. The Command Subsystem uses these samples to verify real-time, load and stored commands.

FOS Telemetry Subsystem: Received Dump notification from FormatCommand.

FOS Resource Management Subsystem: RMS starts the command tasks running as part of a logical string and then supplies EOC spacecraft contact and commanding session configuration information. This information includes command database selection, valid authorized user ID, prerequisite setting, and whether the process is acting as part of the primary or backup command subsystem, and other configuration changes from FUI which are routed through RMS.

FOS Data Management Subsystem: Command database and stored memory load information may be retrieved from the Data Management Subsystem. Information pertaining to user authorization, command bit pattern definition, validation, and verification is stored in the command database. Spacecraft and instrument commands to be executed autonomously, as well as flight software, may be contained within the stored memory loads. The Data Management Subsystem receives, stores, and forwards to the appropriate subsystems and instrument teams, the command event messages (command uplink status, command verification, command notification and command load status) generated by the FormatCommand process. Configuration files, part of the database, are read from the DMS. Snapshot files are written to and read from the DMS The configuration file information includes the max downlink time and the setting of the prerequisite checking (i.e., enabled or disabled). FormatCommand also uses DMS to access Load Catalog Entry. Once a load, or a 4k load partition, is confirmed as uplinked, CMS is notified so that it can track the progress of loads.

FOS User Interface Subsystem: The User Interface delivers command and load directives to the FormatCommand process. The source of individual directives may be EOC automated ground scripts or FOT input that have been parsed by the User Interface prior to delivery to the FormatCommand process. For each command or load directive, the Command Subsystem performs an authorization check and validation. For transmission of critical spacecraft and instrument commands and loads, the Commanding Subsystem prompts FOT for confirmation via the User Interface, and User Interface delivers FOT's response to the Command Subsystem. Similarly, a prerequisite override prompt gives the operator the option to uplink commands that fail prerequisite state checking. The User

Interface also receives and displays command status generated by the Command Subsystem.

FopCommand process: The command, in its 1553-B format, or the packet in CCSDS format, is forwarded to the FopCommand process, where the command or load packet is further prepared for uplinking. Command receipts, which confirm the final uplink status of commands and loads, are received from the FopCommand process.

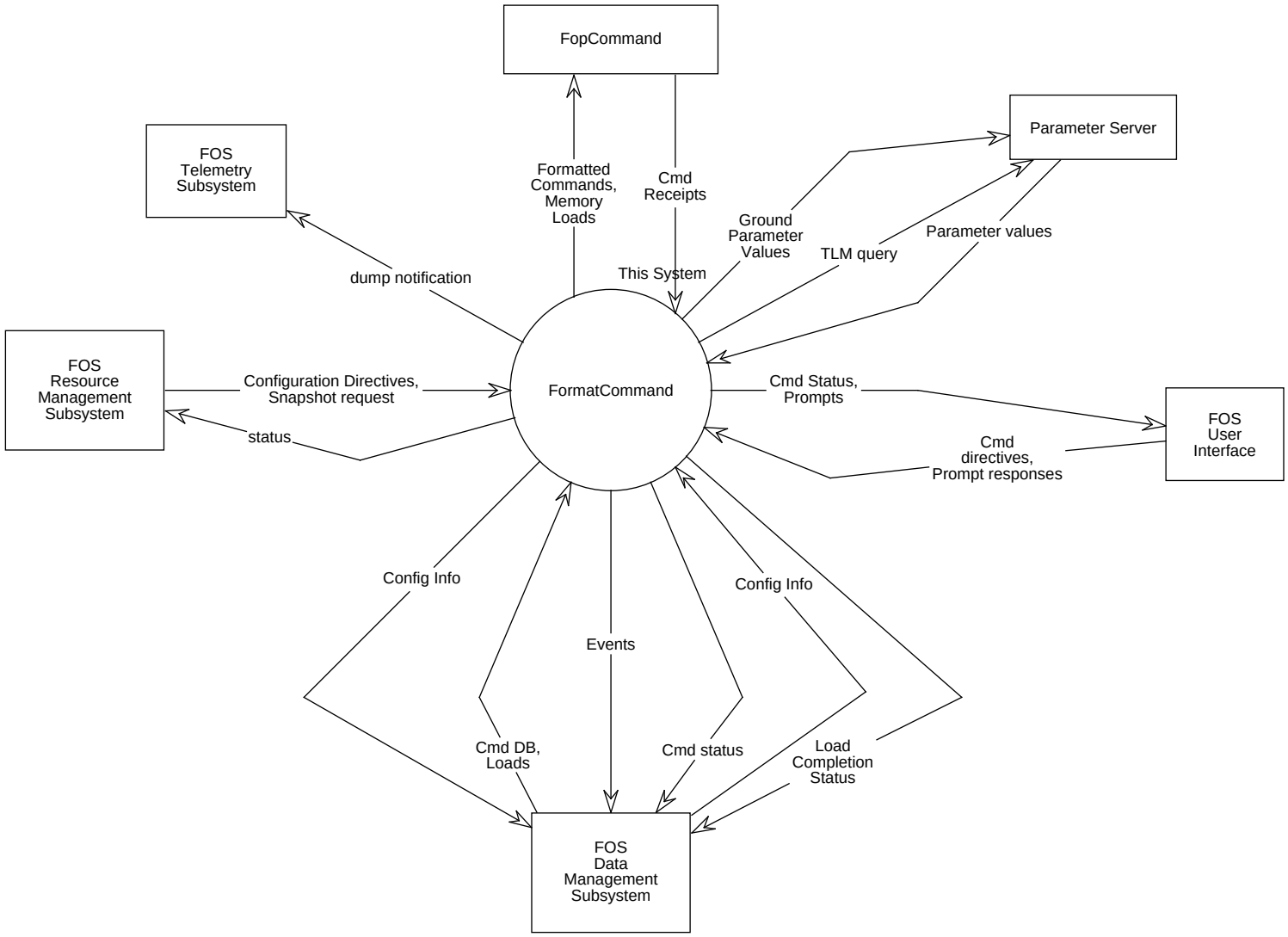

Figure 3.2.1-1. FormatCommand Context Diagram

3.2.2 FormatCommand Interfaces

Table 3.2.2. FormatCommand Interfaces (1 of 4)

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Forwards commands	FcCm CCSDSFop Proxy	Forwards Cmds to FopCommand process	CMD: Fop Command	CMD: Format Command	once per command
	FcGnFop CmdMsg	Message containing a r/t command, in 1553-b format			
	FcGnFop PacketMsg	Message containing a CCSDS packet			
Passing acks and status for commands	FcCdFop Format Proxy	Receives acknow-ledgments & uplink receipts from the FopCommand process	CMD: Format Command	CMD: Fop Command	two messages per command
	FcGnFop AcceptMsg	Acknowledgement message			
	FcGnFop Receipt Msg	Message confirming onboard receipt of a command			
Provide Configur-ation Info	FoCdRms CmdProxy	Sends configuration directives from RMS to FormatCommand process	CMD: Format Command	RMS: String Manager	< 10 per pass, or < 280 / day
	FoGnRms SetPrereq CheckMsg	Message to change prereq check override setting			
	FoGnRms Shutdown Msg	Message notifying FormatCommand process to terminate			
	FoGnRms Save Snapshot Msg	Message requesting Format Command to create a snapshot file			
	FoGnRms SetCmd AuthUser Msg	Message requesting cmd authorization be set to specified user/ workstation			

Table 3.2.2. FormatCommand Interfaces (2 of 4)

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
	FoGnRms Fromat Primary Mode	Message to change the setting of primary mode			
	FoGnRms Format InitMsg	Message containing initialization values			
Returns directive Status	FoGnCmd RmsIf	Returns a completion status of directives to RMS	CMD: Format Command	RMS: String Manager	< 10 per pass, or < 280 / day
	FoGnRms ReceiptMsg	Directive completion message			
	FoGnRms SetCmd AuthUser Msg	Message requesting cmd authorization be set to specified user/ workstation			
	FoGnRms Fromat Primary Mode	Message to change the setting of primary mode			
	FoGnRms Format InitMsg	Message containing initialization values			
Returns directive Status	FoGnCmd RmsIf	Returns a completion status of directives to RMS	CMD: Format Command	RMS: String Manager	< 10 per pass, or < 280 / day
	FoGnRms ReceiptMsg	Directive completion message			
Provide commands & prompt responses	FoGnFui CmdProxy	Sends r/t & stored commands, and prompt responses from FUI to FormatComand process	CMD: Format Command	FUI: Ground Script Controller	1-3 per command
	FoGnFuiRt CmdMsg	Message containing a real time command			
	FoGnFui StoredCmd Msg	Message containing a stored command			

Table 3.2.2. FormatCommand Interfaces (3 of 4)

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
	FoGnPart RspMsg	Message containing user response to a partitioned load error override prompt			
	FoGnFui Critical RspMsg	Message containing user response to a critical prompt			
	FoGnFui Prereq RspMsg	Message containing user response to a prerequisite override prompt			
	FoGnFui LoadMsg	Message instructing Format Command to begin processing a load			
	GoGnFui AbortLoad Msg	Message instructing Format Command is to interrupt the load in progress			
Provide command status info	FoGnCmd FuiIf	Sends command status messages to FUI	CMD: Format Command	FUI: Ground Script Controller	1-5 per command
	FoUi Instruction	Message containing the command status			
Interface to TLM subsystem	FoGnCmd TlmProxy	Sends messages to the TLM subsystem	CMD: Format Command	TLM: Decom	< 1 / day
	FoGnTlm DumpMsg	Msg notifying TLM subsystem that a dump command has been issued			
Provides access to data values	Parameter Server Interface	Sends ground parameters and receives TLM parameters	Parameter Server	CMD: Format Command	nominally < 12 per command

Table 3.2.2. FormatCommand Interfaces (4 of 4)

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
	ClientPid List	List of parameters for which data is to be received			
	ClientBuffer	Data buffer containing S/C parameter values			
	Parameter	Contains Format Command Config values			
	FoUi Instruction	Message containing the command status			
Interface to TLM subsystem	FoGnCmd TlmProxy	Sends messages to the TLM subsystem	CMD: Format Command	TLM: Decom	< 1 / day
	FoGnTlm DumpMsg	Msg notifying TLM subsystem that a dump command has been issued			
Provides access to data values	Parameter Server Interface	Sends ground parameters and receives TLM parameters	Parameter Server	CMD: Format Command	nominally < 12 per command
	ClientPid List	List of parameters for which data is to be received			
	ClientBuffer	Data buffer containing S/C parameter values			
	Parameter	Contains Format Command Config values			
Event Logging	FdEvEvent Logger	Provides routing and archiving of events messages	DMS: FdEvEvent Archiver	CMD: Format Command	1-5 per command

3.2.3 FormatCommand Object Model Description

The design scope for the FormatCommand process Object Model (Figure 3.2.3-1) is the commanding of a single EOS spacecraft and its instruments via a single logical string.

The FcCdCmdController class controls the flow of operations to process commands into the 1553-B spacecraft format. It is responsible for initialization of the FormatCommand process and is the entry point for command and load directives. Command directives are received via the FcCdReceiveIf class from the User Interface Subsystem for processing by the FcCdCmdController. The FcCdCmdController's role is to coordinate validation, build, transmission, receipt verification and execution verification for real-time commands and loads and to coordinate execution verification alone for stored commands. For real-time commands, it is responsible for validating the command submitted in the command directive utilizing the command database; initializing the command's verification modes; and initiating building of the command. ASTER command status will be exported to the ASTER control center via the FdEvEventLogger class in addition to the EOC via the User Interface Subsystem. Note that commands processed by FcCdCmdController may be in various formats; e.g., 1553-B (AM-1 real-time) and CCSDS packets (load data). For stored commands (commands that have previously been loaded and are currently executing onboard the spacecraft), real-time commands (which are executed upon receipt onboard the spacecraft), and loads, it retrieves the telemetry verification point information from the command database, creates the FcCdCmd objects with the verification point information embedded within them, and then initiates the verification process. Upon execution verification, the (stored or real-time) command verification status is recorded for each command directive and displayed by User Interface.

The FcCdCmdQueue is used to keep a record of commands while they are waiting for receipt verification and telemetry verification. It contains two data structures: a linked list of commands which can be quickly referenced by their sequence number, and a hash table of parameters, each of which has its own linked list of commands which verify using that parameter. Thus, when an updated parameter value comes in, all the commands which verify using that parameter are readily accessed. The queue itself has operations to add and remove commands, to begin tlm updates for a command, to verify commands, and to identify time-out commands.

The FoGnRmsCmdProxy is used by RMS to configure FormatCommand

The FoGnFuiCmdProxy is used by FUI to send commands, loads, prompt responses, and load aborts to FormatCommand.

The FoGnCmdFuiIf is used to send command status messages to FUI.

The FoGnFormatTlmIf is used to send messages to TLM.

The FoGnCmdTlmProxy is used to send parameter service requests to TLM Parameter Server.

The FcCdFopFormatProxy is used by FopCommand process to send accept and receipt status (for a real time command or a memory load packet) to FormatCommand.

The FoPsClientIF is a proxy for parameter server. It is used by FormatCommand to serve parameters.

The FoGnGenericMessage class (Figure 3.2.3-2) is a base class that represents all messages received by FormatCommand, and some messages sent by FormatCommand. These messages are not listed individually in this description. They can be found in a separate object model and in the

interface table.

The FoUiStatus class is sent to FUI to inform it of command status.

The FcCdCommandDatabase class is a container class that contains instances of the FcCdCmdDef class for a single spacecraft and its instruments. The FcCdCmdDef class contains a single command definition. The FcCdCommandDatabase is initialized from the Data Management Subsystem via the FoDsFile class. Command definitions are templates for spacecraft and instrument commands and are used in the validation of command directives.

The FcCdBaseCmd class is the base class for all command classes. It contains addresses of proxies and interfaces used by all command classes.

The FcCdHexCmd class contains the hex/binary command in 1553-B format.

The FcCdCmd class specifies the common attributes and operations for command classes within the Command Subsystem. FcCdCmd is further specified as three subclasses, FcCdRtCmd, FcCdStoredCmd and FcCdLoadCmd.

The FcCdStoredCmd is used for telemetry verified of stored command. It contains the maximum down link time to accommodate for the delay.

The FcCdRtCmd class is responsible for building the binary bit pattern for uplink to the spacecraft. The FcCdRtCmd class does prerequisite checking and execution verification of commands uplinked to the spacecraft by accessing telemetry values via the FoGnCmdTlmProxy. A subclass of the FcCdRtCmd class is the FcCdLoadCmd class. The FcCdLoadCmd class handles spacecraft and instrument memory loads.

The FcCdLoadCmd coordinates load processing by accessing and forwarding load data in the FcCdLoadData object, in CCSDS packet format, from FcCdLoadCmd to FcCdCmdController. The FcCdLoadCmd class verifies loads uplinked to the spacecraft by accessing telemetry values via the FoGnCmdTlmProxy class.

The FcCdLoadData class holds the data for a load.

The single instantiation of the FcCdCmdQueue class holds the commands and loads waiting for telemetry verification. It stores them in two member object data structures, a linked list of commands and loads, with the verification parameter of each; and a hash table which uses the telemetry parameters as keys, with pointers to all the commands verifying off of each parameter.

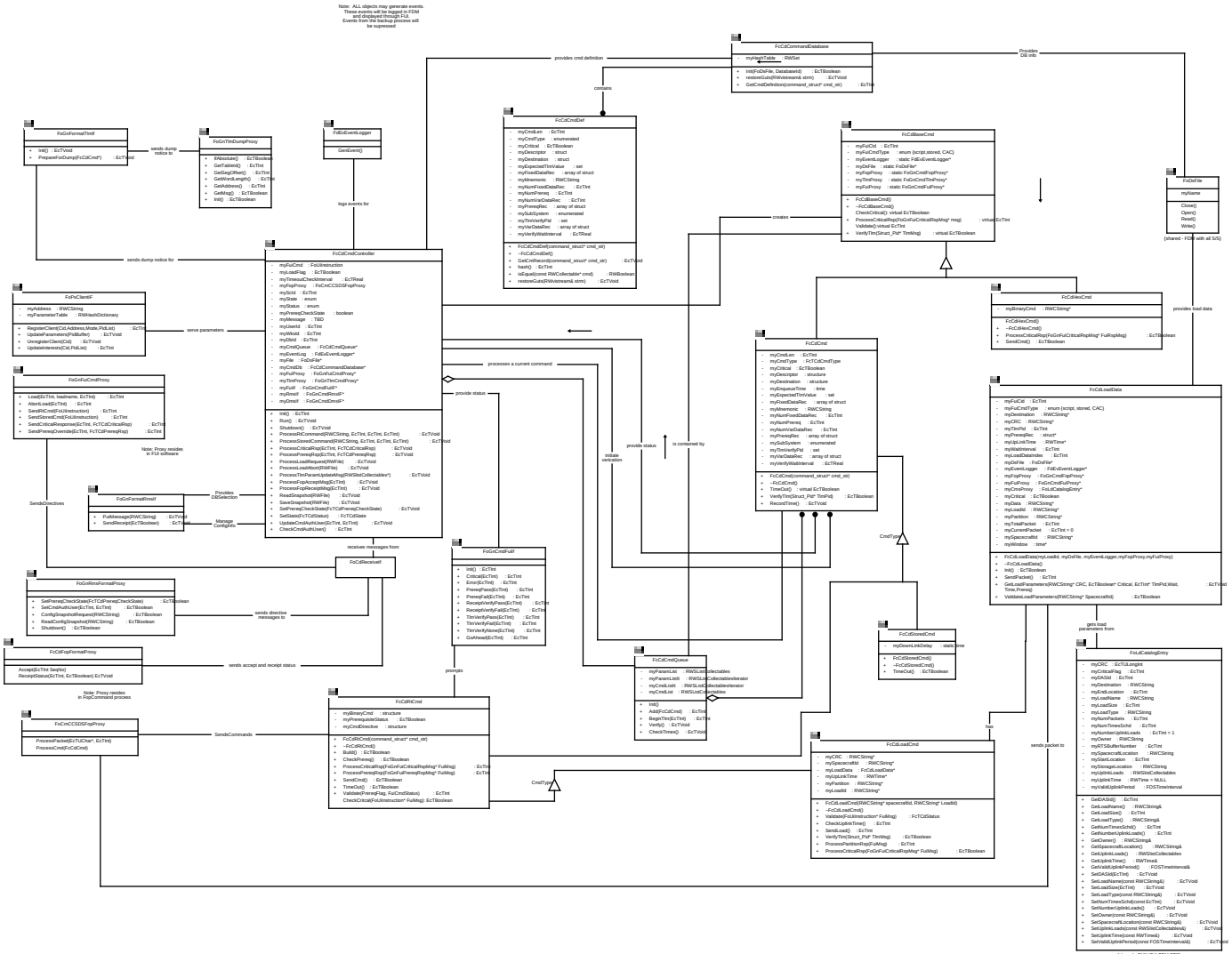

Figure 3.2.3-1. FormatCommand Object Diagram

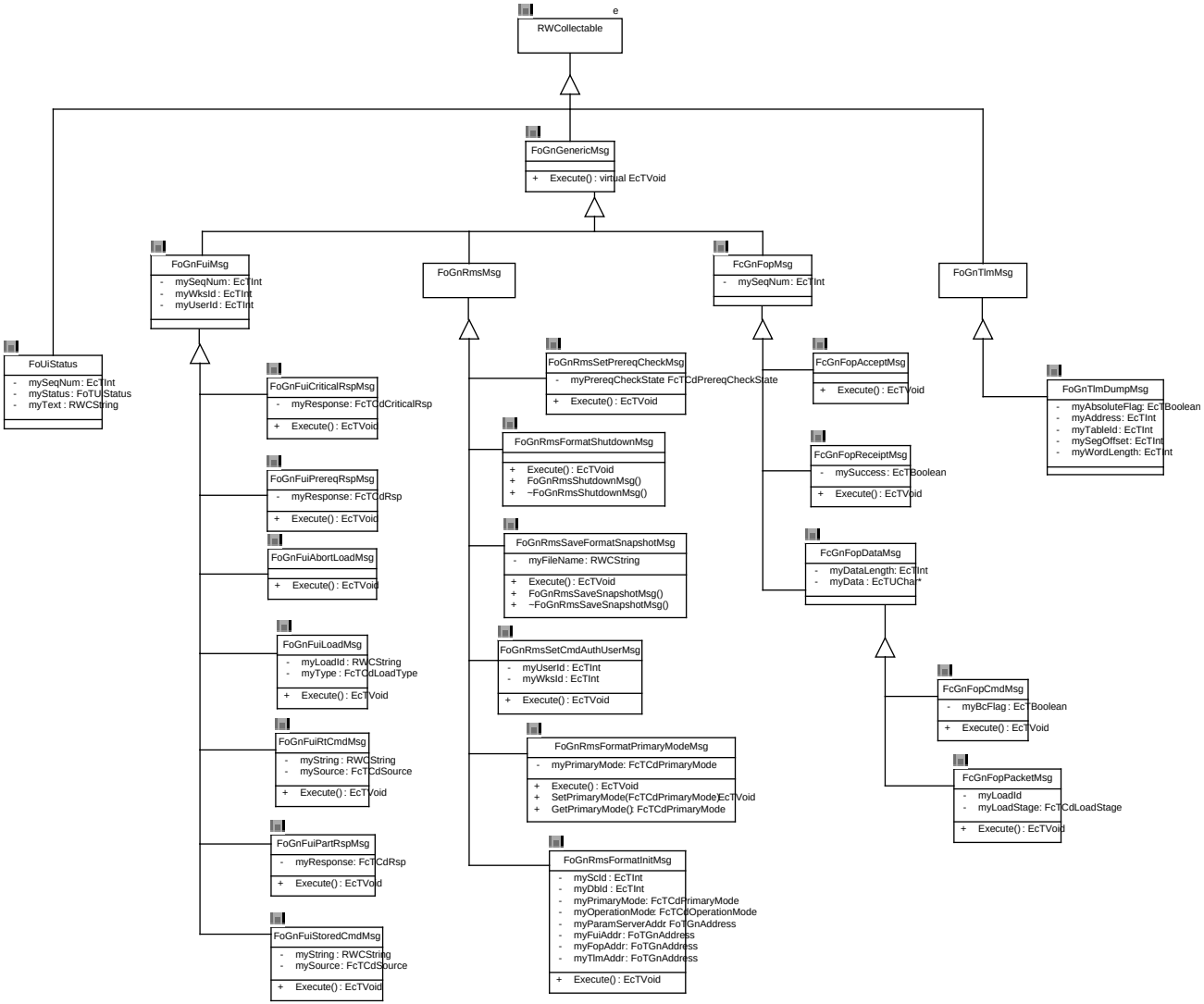

Figure 3.2.3-2. FormatCommand Message Object Diagram

3.2.4 FormatCommand Subsystem Dynamic Model

The following are the FormatCommand Subsystem scenarios which are defined in this section.

Real-Time Command: Initialization for Primary Process
Real-Time Command: Initialization for Back Up Process
Real-Time Command: Change User Authorization
Real-Time Command Validation: Successful
Real-Time Command Validation: No Command Definition
Real-Time Command Validation: Fail Submnemonic Check
Real-Time Command Validation: Fail Due to No Override
Real-Time Command Validation: Fail Due to Cancel Critical
Stored Command Validation
Stored Command Validation - No Verification
Write Configuration Snapshot Request
Read Configuration Snapshot Request
Load Command Validation: Successful
Load Command Validation: Fail Due to Missing Load
Load Command Validation: Fail Due to Invalid Parameter
Load Command Validation: Fail Due to Cancel Critical
Load Command Validation: Abort
Hex Command Validation: Success
Hex Command Validation: Fail Due to Cancel Critical
Real-Time Command Verification: Success
Real-Time Command Verification: Failure Due to Timeout
Real-Time Load Verification: Success
Real-Time Load Verification: Failure Due to Timeout
Real-Time Dump Command

Additionally, state diagrams for the Command Controller (FcCdCmdController), the Real-Time Command (FcCdRtCmd), and the Load Command (FcCdLoadCmd) objects are included.

3.2.4.1 Real-Time Command FormatCommand Initialization: Successful Scenario for Primary Process

3.2.4.1.1 Real-Time Command FormatCommand Initialization: Successful for Primary Process Abstract

The purpose of the "Real-Time Command FormatCommand Initialization: Successful for Primary Process" scenario is to describe the process by which the FormatCommand software of the FormatCommand process is initialized.

Figure 3.2.4.1-1 is the event trace diagram which corresponds to this scenario.

3.2.4.1.2 Real-Time Command FormatCommand Initialization: Successful Summary Information

Interfaces:

Parameter Server

Data Management Subsystem

Resource Management Subsystem

FormatCommand

FOS User Interface

Stimulus:

The Resource Manager (RMS) starts up the FormatCommand process.

Desired Response:

The Resource Manager receives the status of successful FormatCommand initialization.

Pre-Conditions:

Configuration file must be identified and available.

Post-Conditions:

The FormatCommand is placed in the "wait" state, and ready for directives.

3.2.4.1.3 Scenario Description

The main operation of the FormatCommand application (FcCmFopAppl) is invoked when the Resource Manager (RMS) starts up the process. The command line will contain the IPC address of the RMS. This address is forwarded to the FcCdCmdController, the controller of the FormatCommand processing. The IPC address is used to establish communication with the RMS, via FoGnFormatRmsIf. Once communication is established, the process waits for an initialization message from RMS. Upon receipt of the message, the message is read and input parameters are extracted from the message. These parameters contain IPC addresses which are used to establish communications with other processes, specifically DMS, TLM, Parameter Server, FOS User Interface and the FopCommand process. Other parameters include the spacecraft ID, database ID, and the process "role" as part of a primary string.

A DMS connection is established via FdEvEventLogger for events processing.

FoDsFile is then utilized to access the database file. The file information is used to configure the FormatCommand attributes and will contain default values for various attributes. This configuration information is then used to configure the FormatCommand.

Then several objects which will exist for the life of the string are instantiated.

A successful completion status is returned to RMS and a "successful initialization" event message is logged via FdEvEventLogger.

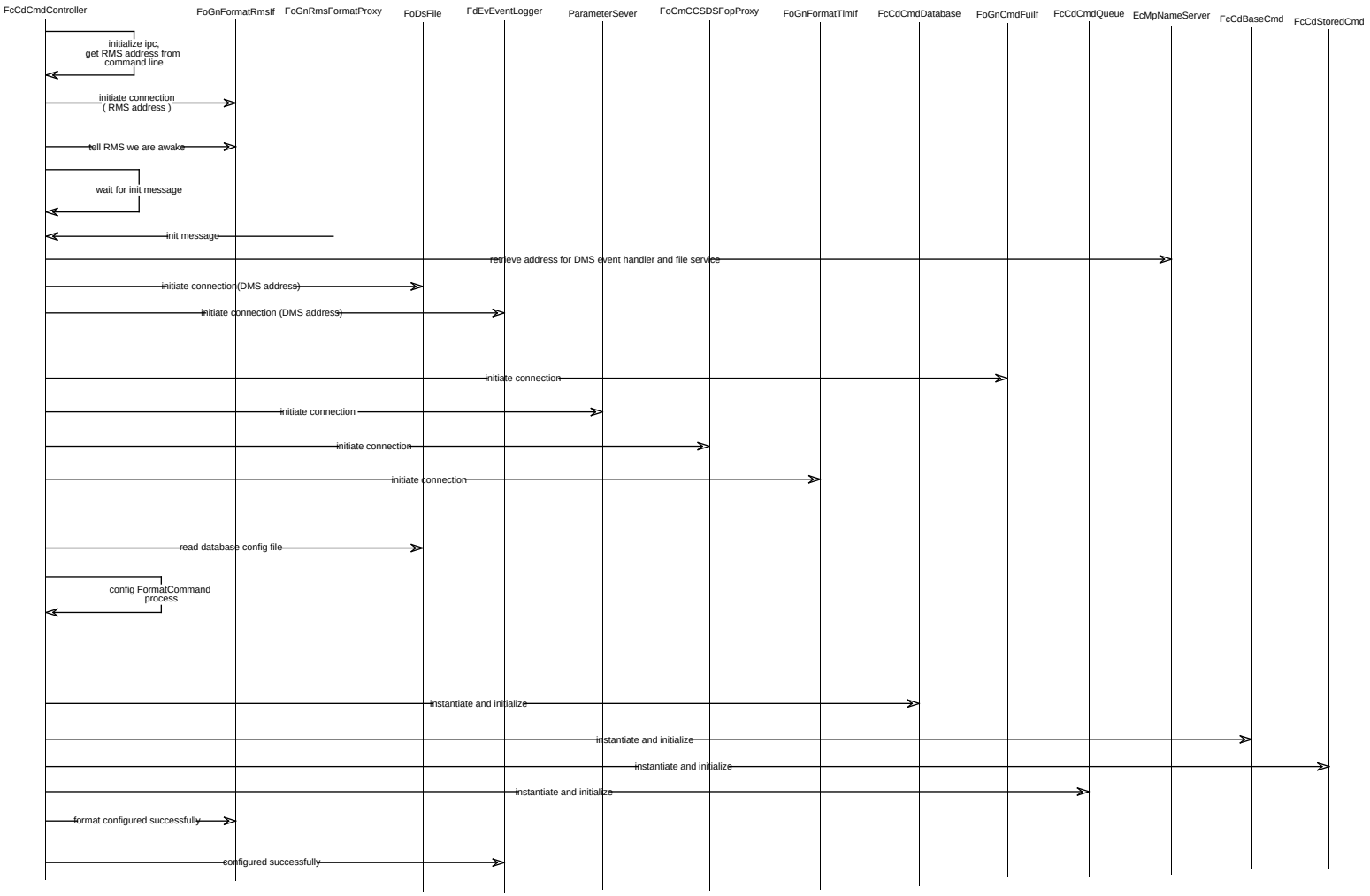

Figure 3.2.4.1-1. FormatCommand Initialization: Successful for Primary Process

3.2.4.2 Real-Time Command FormatCommand Initialization: Successful Scenario for Back Up Process

3.2.4.2.1 Real-Time Command FormatCommand Initialization: Successful for Back Up Process Abstract

The purpose of the "Real-Time Command FormatCommand Initialization: Successful for Back Up Process" scenario is to describe the process by which the FormatCommand software of the FormatCommand process is initialized.

Figure 3.2.4.2-1 is the event trace diagram which corresponds to this scenario.

3.2.4.2.2 Real-Time Command FormatCommand Initialization: Successful Summary Information

Interfaces:

Parameter Server

Data Management Subsystem

Resource Management Subsystem

FormatCommand

FOS User Interface

Stimulus:

The Resource Manager (RMS) starts up the FormatCommand process.

Desired Response:

The Resource Manager receives the status of successful FormatCommand initialization.

Pre-Conditions:

Configuration file must be identified and available.

Post-Conditions:

The FormatCommand is placed in the "wait" state, and ready for directives.

3.2.4.2.3 Scenario Description

The main operation of the FormatCommand application (FcCmFopAppl) is invoked when the Resource Manager (RMS) starts up the process. The command line will contain the IPC address of the RMS. This address is forwarded to the FcCdCmdController, the controller of the FormatCommand processing. The IPC address is used to establish communication with the RMS, via FoGnFormatRmsIf. Once communication is established, the process waits for an initialization message from RMS. Upon receipt of the message, the message is read and input parameters are extracted from the message. These parameters contain IPC addresses which are used to establish communications with other processes, specifically DMS, TLM, Parameter Server, FOS User Interface and the FopCommand process. Other parameters include the spacecraft ID, database ID, and the process "role" as part of a backup string.

A DMS connection is established via FdEvEventLogger for events processing.

FoDsFile is then utilized to access the database file. The file information is used to configure the FormatCommand attributes and will contain default values for various attributes. This configuration information is then used to configure the FormatCommand.

Because it is a backup process, FcCdCmdController reads the snapshot file and does further configuration based on the values in it.

Then several objects which will exist for the life of the string are instantiated.

A successful completion status is returned to RMS and a "successful initialization" event message is logged via FdEvEventLogger.

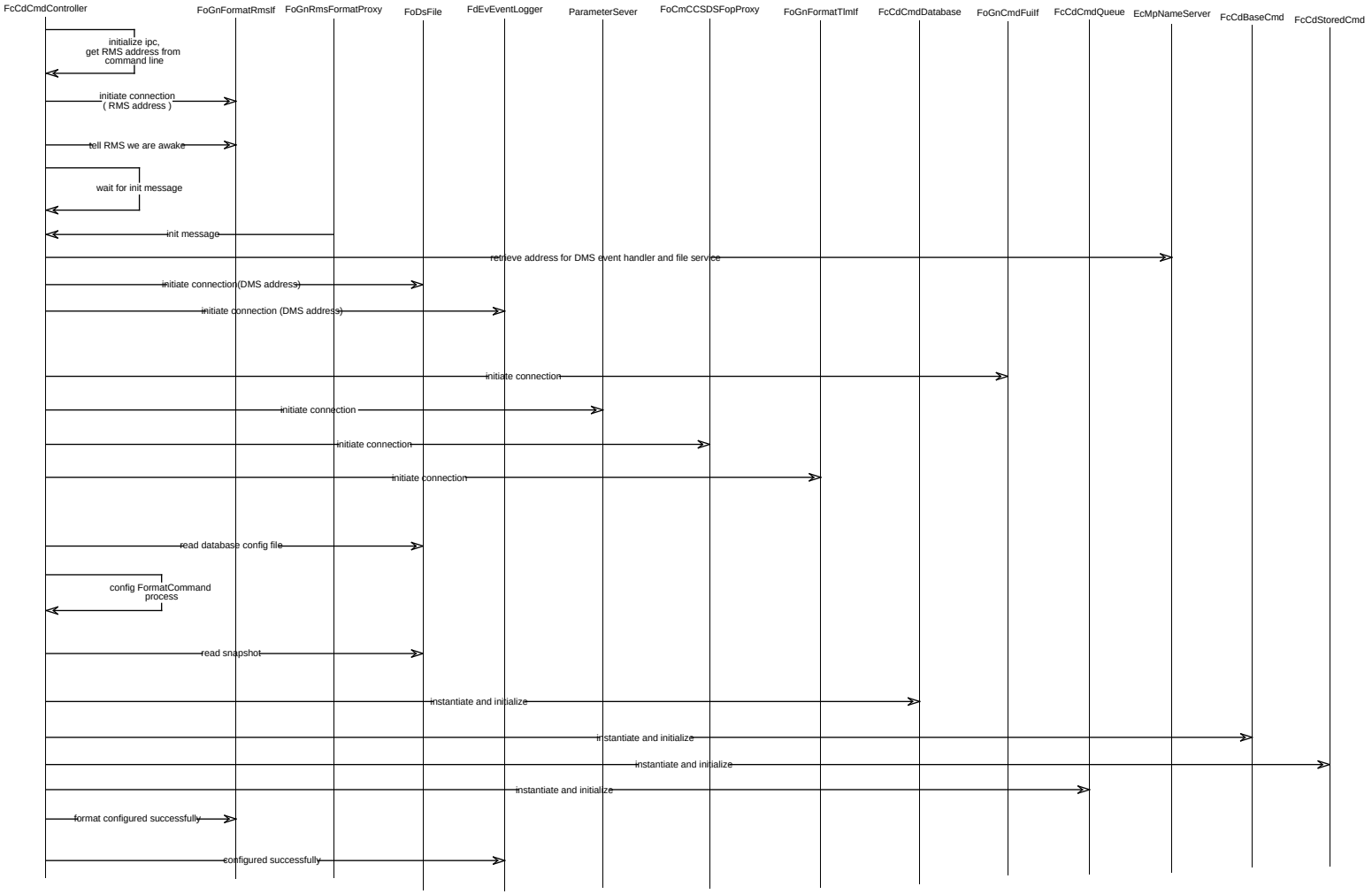

Figure 3.2.4.2-1. FormatCommand Initialization: Successful for Back Up Process

3.2.4.3 Real-Time Command FormatCommand Change Authorized User: Successful Scenario

3.2.4.3.1 Real-Time Command Format Command Change Authorized User: Successful Abstract

The purpose of the "Real-Time Command FormatCommand Change Authorized User: Successful" scenario is to describe the process by which the FormatCommand software of the FormatCommand process is directed to change its authorized user..

Figure 3.2.4.3-1 is the event trace diagram which corresponds to this scenario.

3.2.4.3.2 Real-Time Command FormatCommand Change Authorized User: Successful Summary Information

Interfaces:

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) sends a message directing FormatCommand to change its authorized user..

Desired Response:

The Resource Manager receives the status of successful execution of the directive.

Pre-Conditions:

Communications established with RMS.

Post-Conditions:

The authorized user has been changed.

3.2.4.3.3 Scenario Description

RMS sends a directive message via the FoGnRmsFormatProxy to the FcCdCmdController, which changes the authorized user information and has FoGnFormatRmsIf send a receipt message to RMS indicating success.

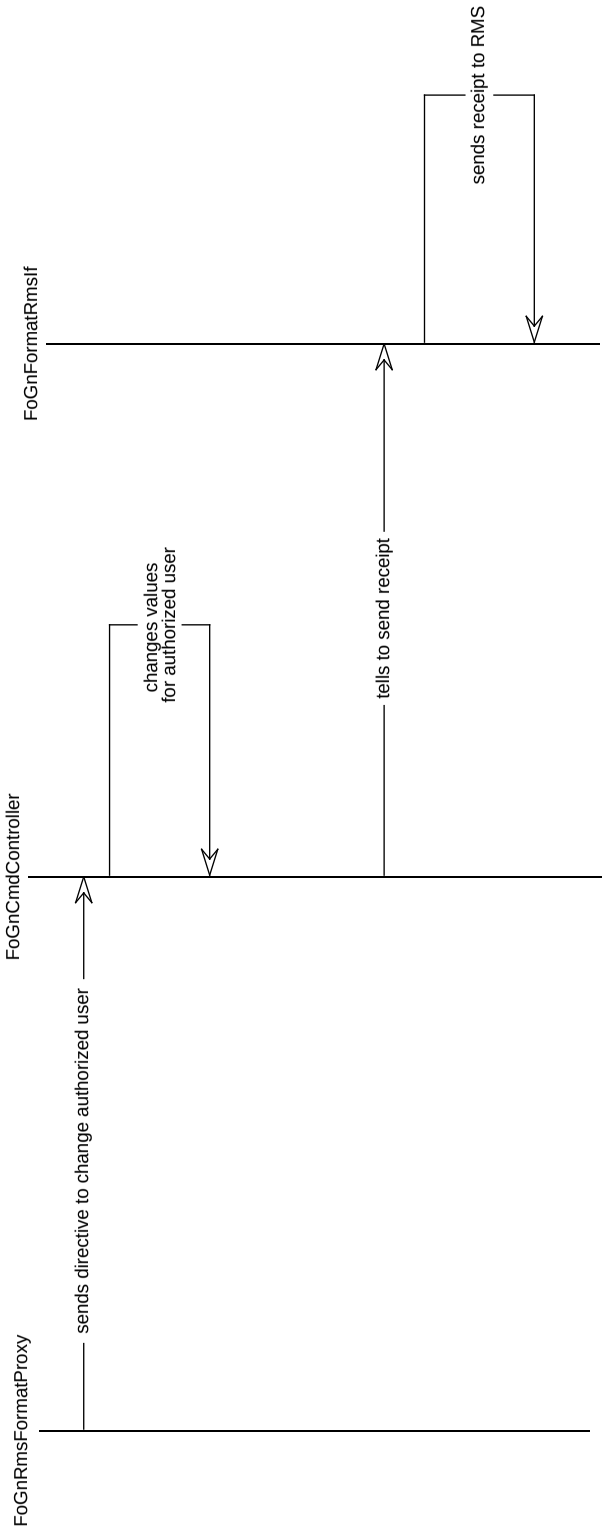

Figure 3.2.4.3-1. FormatCommand Change Authorized User: Successful

3.2.4.4 Real-Time Command Validation: Successful Scenario

3.2.4.4.1 Real-Time Command Validation: Successful Abstract

The purpose of the "Real-Time Command Validation: Successful" scenario is to describe the process by which a real-time command is validated and built. The validation check is done before the command is actually built (translated from a mnemonic to a string of bits).

Figure 3.2.4.4-1 is the event trace diagram which corresponds to this scenario.

3.2.4.4.2 Real-Time Command Validation: Successful Summary Information

Interfaces:

FOS User Interface

Telemetry Subsystem

Data Management Subsystem

UplinkCommand Process

Stimulus:

A command directive is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of successful command validation/generation.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

The command is assembled into the command protocol format. This format is specified as 1553 bus for AM-1.

3.2.4.4.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a real-time command, in mnemonic format. The command in the scenario is a critical command for the ASTER instrument.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded. The query also reveals that: 1) the command is not a load command, and 2) the command is for the ASTER instrument. FdEvEventLogger echos the command, and it is eventually forwarded (by DMS) to the ASTER control center (as a notification message).

FcCdCmdController then uses the information from the database query in the creation of an FcCdRtCmd object. During its creation, the FcCdRtCmd object is imbedded with information

supplied by the FcCdCommandDatabase information (such as the binary command, verification information, prerequisite states, criticality) as well as with information about the user who entered the command. Thus, the object contains all the information it needs to do validation.

FcCdCmdController then invokes the Validate operation of the FcCdRtCmd, which performs the following:

It does syntax checking of the command directive as items such as submnemonics (required and optional; database defined default values are substituted for omitted optional submnemonics) are validated.

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). The CheckPrereq operation performs this function. The prerequisite check is positive if the telemetry points are active (recently updated) and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessPrereqRsp operation. The user's response is to allow; the proper notifications are made via FdEvEventLogger and validation processing continues.

It issues a critical prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessCriticalRsp operation. The user's response is to allow, the proper notifications are made via FdEvEventLogger and control returns to FcCdCmdController.

The command is now successfully validated and ready to be built.

FcCdCmdController invokes the Build operation of the FcCdRtCmd object, the end result of which is a command built according to the command protocol: 1553 bus.

The completed command consists of a command destination, a command descriptor and, optionally, the command data.

The command destination consists of the following:

a database defined Remote Terminal ID, which is supplied to the object upon creation

a database defined Subaddress, also supplied to object upon creation.

a Word Count, which is derived from the length of the command data.

The Command Descriptor consists of the following:

a database defined Command Type, which is supplied to the object upon creation.

a Word count, which is derived from the length of the command data (serial commands only)

a database defined Board address, which is supplied to the object upon creation.

a Channel select, which defines the BDU to be used

a database defined Command Channel, which is supplied to the object upon creation.

The Command data:

The database defines whether or not the data will be present, and if present, the size.

The values of the data (if present) are either:

1) database defined, and supplied to the object upon creation

2) defined by the user, through submnemonic specification

3) have default values which are database defined, and overridable by the user through submnemonic specification

The command is now successfully generated, and an acknowledgment messages is returned to user interface. The command is forwarded to the UplinkCommand process via FcCmCCSDSFop-Proxy, and then added to the queue.

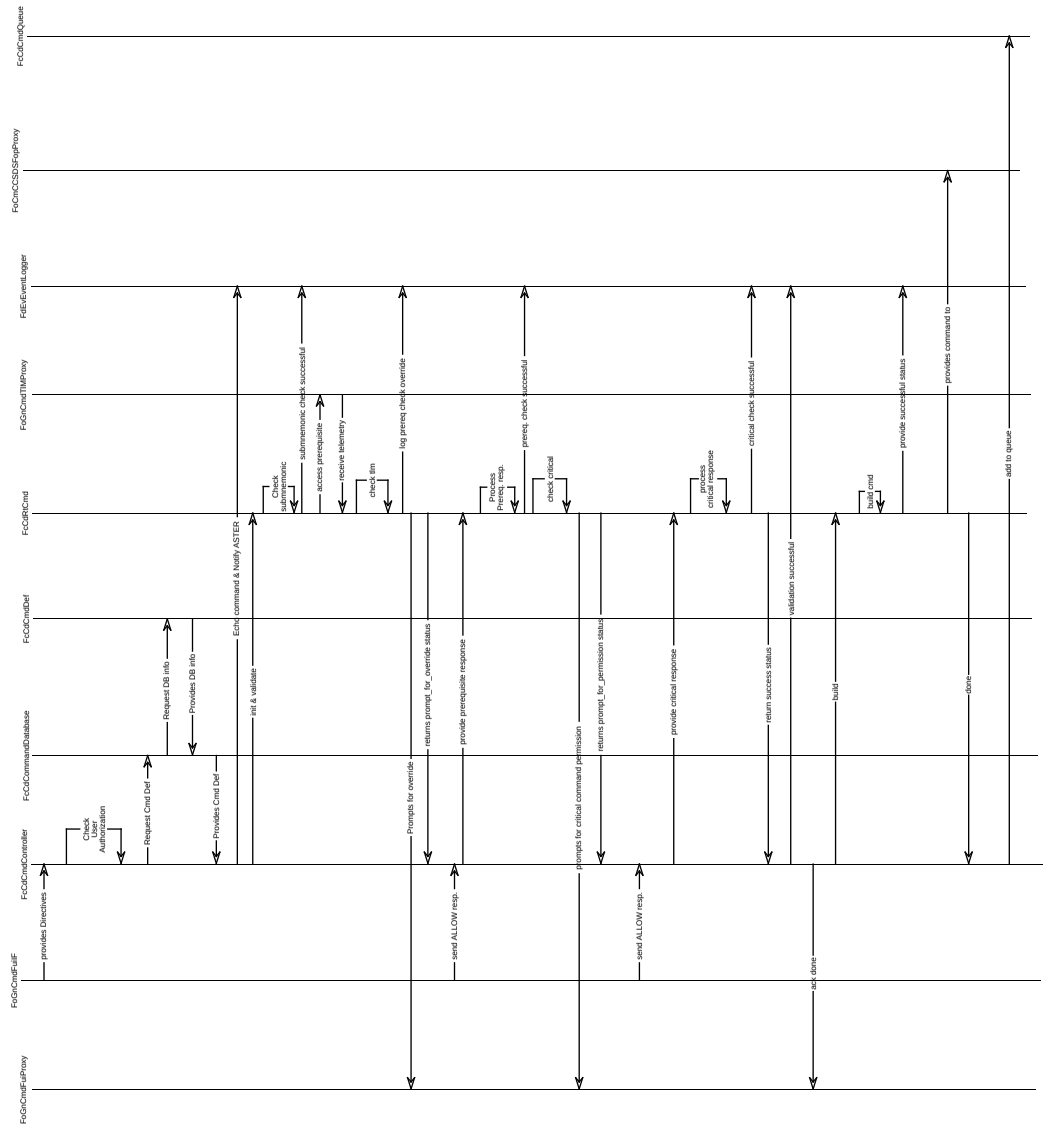

Figure 3.2.4.4-1. Real-Time Command Validation: Successful Event Trace

3.2.4.5 Real-Time Command Validation: No Command Definition Scenario

3.2.4.5.1 Real-Time Command Validation: No Command Definition Abstract

The purpose of the "Real-Time Command Validation: No Command Definition" scenario is to describe the process by which a request to issue an erroneous command is rejected.

Figure 3.2.4.5-1 is the event trace diagram which corresponds to this scenario.

3.2.4.5.2 Real-Time Command Validation: No Command Definition Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

Stimulus:

A erroneous command directive is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of the failed command.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.5.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a real-time command in mnemonic format.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command, but the command mnemonic is not found. FcCdCmdController then echoes a command/error message and returns to User Interface the failure status.

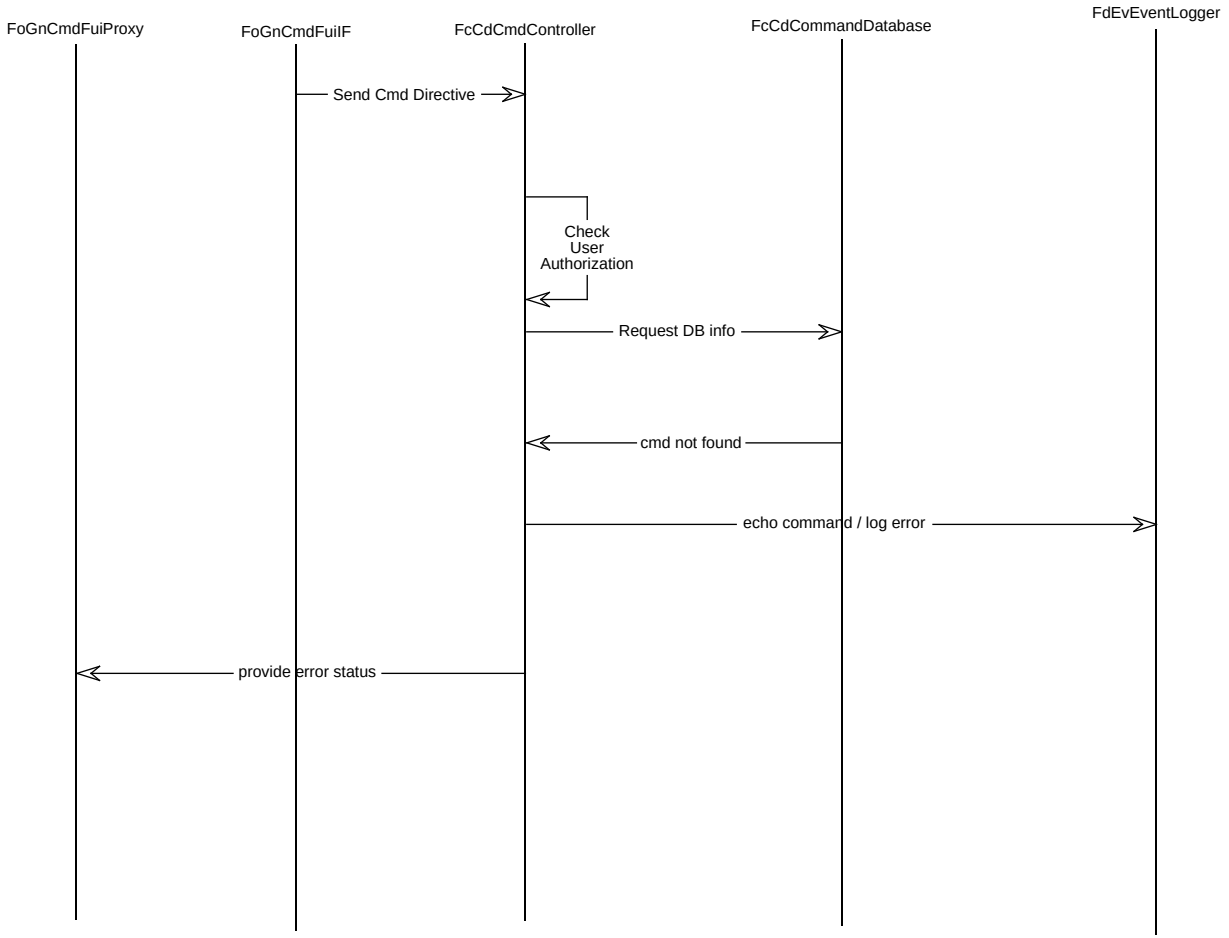

Figure 3.2.4.5-1. Real Time Command Validation: No command definition

3.2.4.6 Real-Time Command Validation: Fail Submnemonic Check Scenario

3.2.4.6.1 Real-Time Command Validation: Fail Submnemonic Check Abstract

The purpose of the "Real-Time Command Validation: Fail Submnemonic Check" scenario is to describe the process by which a real-time command with a error in a submnemonic specification is rejected.

Figure 3.2.4.6-1 is the event trace diagram which corresponds to this scenario.

3.2.4.6.2 Real-Time Command Validation: Fail Submnemonic Check Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

Stimulus:

A command directive with a submnemonic error is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of the failed command.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.6.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a real-time command in mnemonic format. The command in the scenario is a critical command for the ASTER instrument.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded. The query also reveals that: 1) the command is not a load command and 2) the command for the ASTER instrument. FdEvEventLogger echoes the command, and it is eventually forwarded (by DMS) to the ASTER control center (as a notification message).

FcCdCmdController then uses the information from the database query in the creation of an FcCdRtCmd object. During its creation, the FcCdRtCmd object is imbedded with information supplied by the FcCdCommandDatabase information (such as the binary command, verification information, prerequisite states, criticality) as well as with information about the user who entered the command. Thus, the object contains all the information it needs to do validation.

FcCdCmdController then invokes the Validate operation of the FcCdRtCmd, which performs syntax checking of the command directive as items such as submnemonics (required and optional; database defined default values are substituted for omitted optional submnemonics) are validated. During this check, the submnemonic error is detected. FcCdRtCmd issues an error status message to ASTER via FdEvEventLogger, and returns to User Interface the failure status.

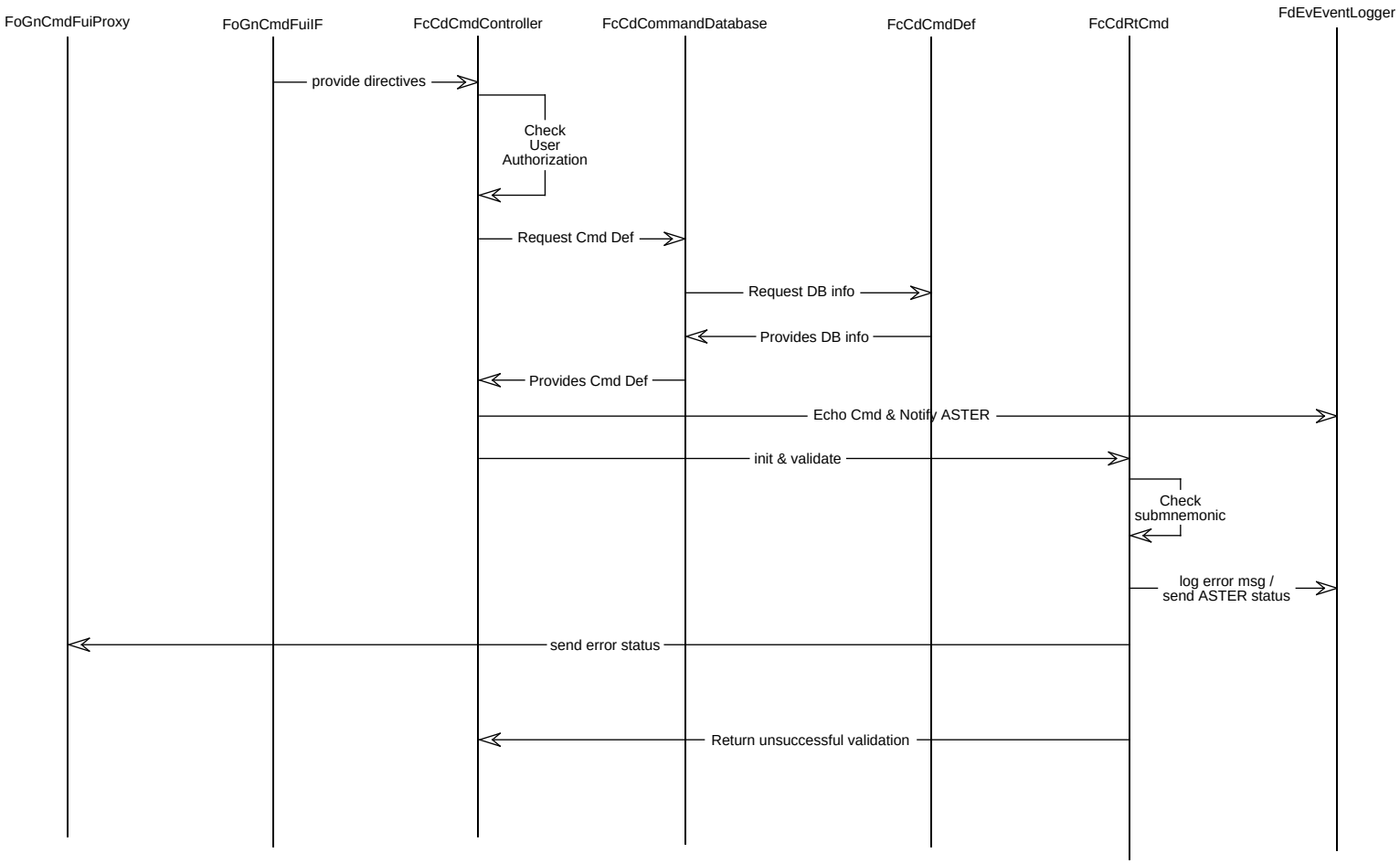

Figure 3.2.4.6-1. Real Time Command Validation: Fail Submnemonic check

3.2.4.7 Real-Time Command Validation: Fail Due to No Override Scenario

3.2.4.7.1 Real-Time Command Validation: Fail Due to No Override Abstract

The purpose of the "Real-Time Command Validation: Fail Due to No Override" scenario is to describe the process by which the processing of a real-time command is terminated once 1) prerequisite checking has failed, and 2) the operator indicates "cancel" to the subsequent prerequisite override prompt.

Figure 3.2.4.7-1 is the event trace diagram which corresponds to this scenario.

3.2.4.7.2 Real-Time Command Validation: Fail Due to No Override Summary Information

Interfaces:

FOS User Interface

Telemetry Subsystem

Data Management Subsystem

Stimulus:

A command directive destined to fail prerequisite state checking is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the completion status of the command.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.7.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a real-time command in mnemonic format. The command in the scenario is a critical command, for the ASTER instrument.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded. The query also reveals that: 1) the command is not a load command and 2) the command for the ASTER instrument. FdEvEventLogger echoes the command, and it is eventually forwarded (by DMS) to the ASTER control center (as a notification message).

FcCdCmdController then uses the information from the database query in the creation of an FcCdRtCmd object. During its creation, the FcCdRtCmd object is imbedded with information

supplied by the FcCdCommandDatabase information (such as the binary command, verification information, prerequisite states, criticality) as well as with information about the user who entered command. Thus, the object contains all the information it needs to do validation.

FcCdCmdController then invokes the Validate operation of the FcCdRtCmd which performs the following:

It does syntax checking of the command directive as items such as submnemonics (required and optional; database defined default values are substituted for omitted optional submnemonics) are validated.

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). The CheckPrereq operation performs this function. The prerequisite check is positive if the telemetry points are active (recently updated) and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessPrereqRsp operation. The user's response is to cancel. This response is forwarded to the FcCdRtCmd object which, via FdEvEventLogger, logs the error message and notifies the ASTER ICC of the completion status. FcCdRtCmd then returns to User Interface the command acknowledgment.

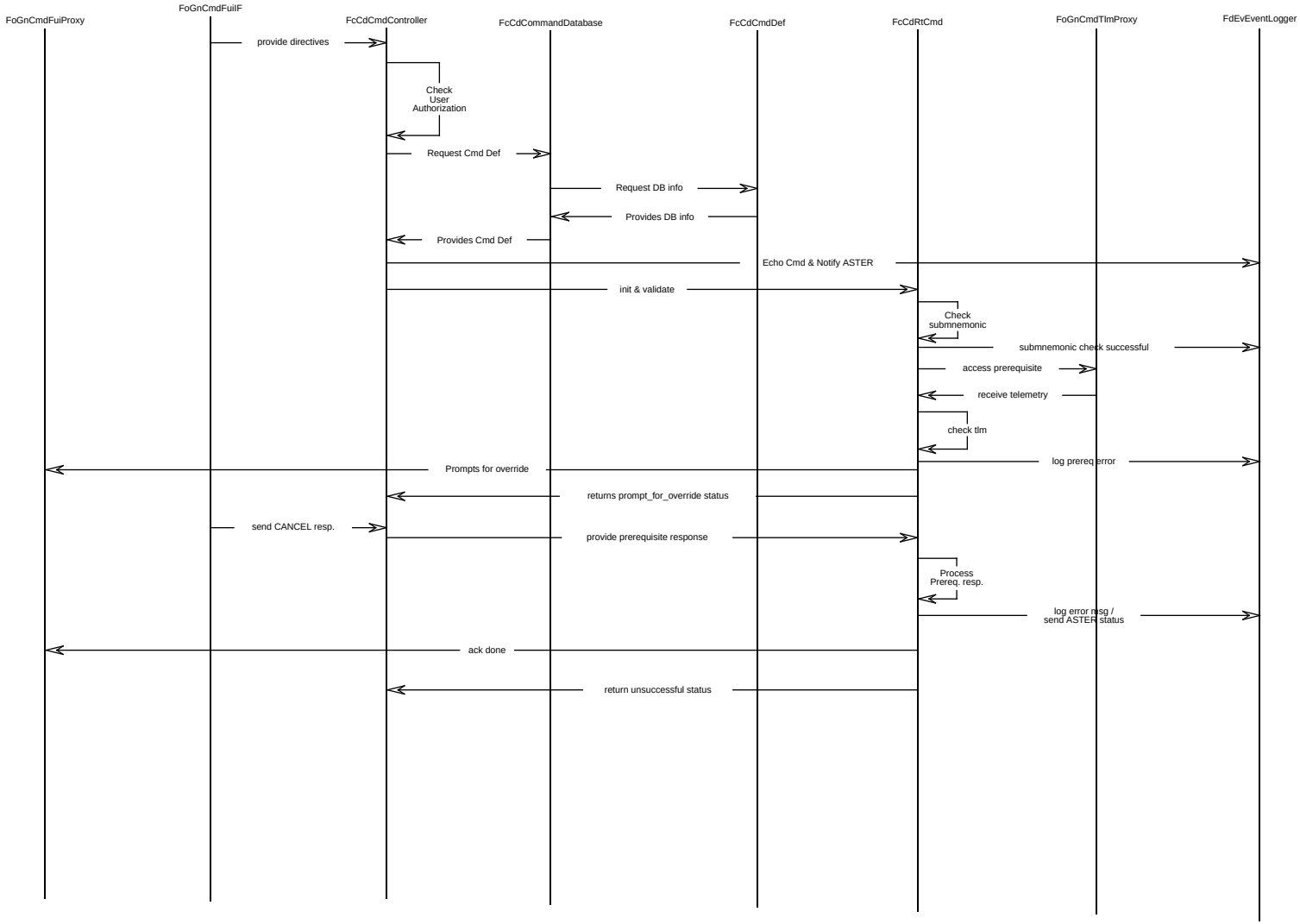

Figure 3.2.4.7-1. Real Time Command Validation: No Prerequisite override

3.2.4.8 Real-Time Command Validation: Fail Due to Cancel Critical Scenario

3.2.4.8.1 Real-Time Command Validation: Fail Due to Cancel Critical Abstract

The purpose of the "Real-Time Command Validation: Fail Due to Cancel Critical" scenario is to describe the process by which a critical real-time command is terminated when the operator indicates "cancel" to the critical prompt.

Figure 3.2.4.8-1 is the event trace diagram which corresponds to this scenario.

3.2.4.8.2 Real-Time Command Validation: Fail Due to Cancel Critical Summary Information

Interfaces:

FOS User Interface

Telemetry Subsystem

Data Management Subsystem

Stimulus:

A command directive for a critical command is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of successful command validation/generation.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.8.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a real-time command in mnemonic format. The command in the scenario is a critical command for the ASTER instrument.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded. The query also reveals that: 1) the command is not a load command, and 2) the command for the ASTER instrument. FdEvEventLogger echoes the command, and it is eventually forwarded (by DMS) to the ASTER control center (as a notification message).

FcCdCmdController then uses the information from the database query in the creation of an FcCdRtCmd object. During its creation, the FcCdRtCmd object is imbedded with information supplied by the FcCdCommandDatabase information (such as the binary command, verification

information, prerequisite states, criticality) as well as with information about the user who entered command. Thus, the object contains all the information it needs to do validation.

FcCdCmdController then invokes the Validate operation of the FcCdRtCmd which performs the following:

It does syntax checking of the command directive as items such as submnemonics (required and optional; database defined default values are substituted for omitted optional submnemonics) are validated.

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). (The CheckPrereq operation performs this function.) The prerequisite check is positive if the telemetry points are active (recently updated), and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessPrereqRsp operation. The user's response is to allow; the proper notifications are made via FdEvEventLogger and validation processing continues.

It issues a critical prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessCriticalRsp operation. The user's response is to cancel. This response is forwarded to the FcCdRtCmd object which, via FdEvEventLogger, logs the error message and notifies the ASTER ICC of the completion status. FcCdRtCmd then returns to User Interface the command acknowledgment.

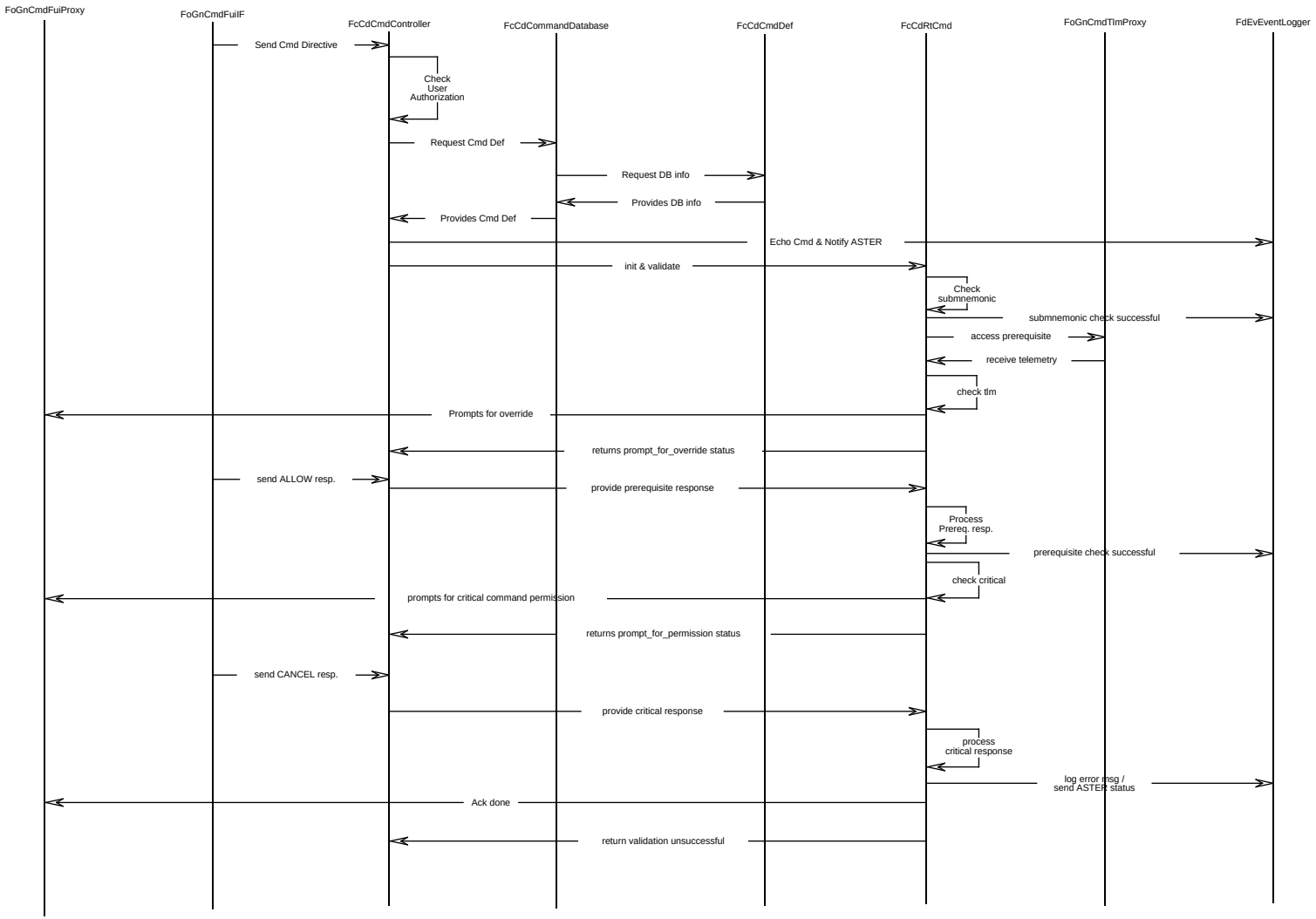

Figure 3.2.4.8-1. Real Time Command Validation: Cancel critical

3.2.4.9 Stored Command Validation Scenario

3.2.4.9.1 Stored Command Validation Abstract

The purpose of the "Stored Command Validation" scenario is to describe the process by which a previously uplinked stored command is validated.

Figure 3.2.4.9-1 is the event trace diagram which corresponds to this scenario.

3.2.4.9.2 Stored Command Validation Summary Information

Interfaces:

FOS User Interface

Telemetry Subsystem

Data Management Subsystem

Stimulus:

A previously uplinked stored command is forwarded by FOS User Interface Subsystem.

Desired Response:

The stored command is successfully validated.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.9.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a previously uplinked, stored command in mnemonic format.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded.

FcCdCmdController then uses the information from the database query to create an FcCdCmd object. During its creation, the FcCdCmd object is imbedded with verification information supplied by the FcCdCommandDatabase.

FcCdCmdController then adds the FcCdCmd object onto the FcCdCmdQueue to be verified.

The command is now successfully validated. FcCdCmdController invokes the PutResponse function of the FoGnCmdFuiProxy to return an acknowledgment to user interface.

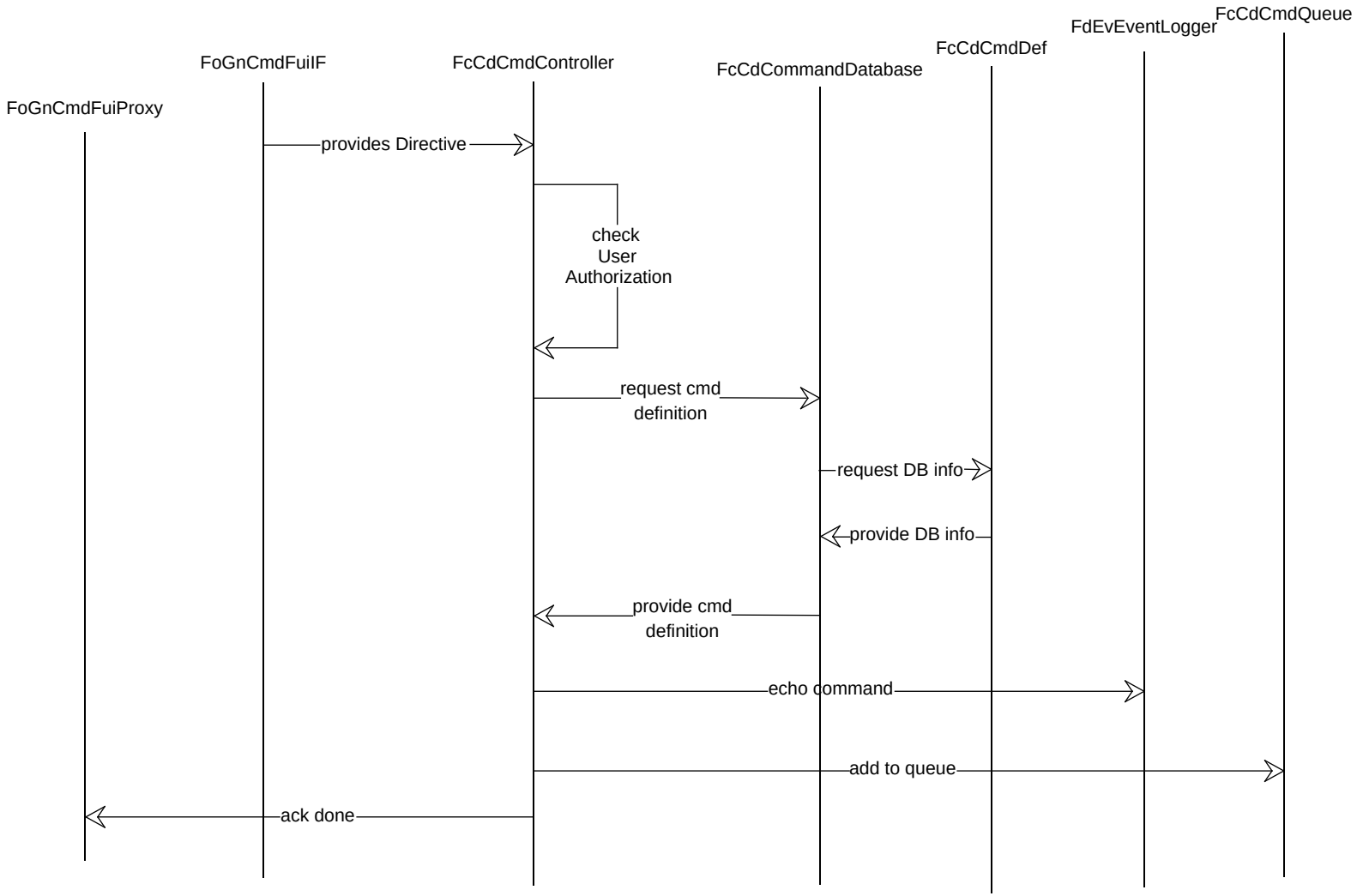

Figure 3.2.4.9-1. Stored Command Validation: Verification required

3.2.4.10 Stored Command Validation - No Verification Scenario

3.2.4.10.1 Stored Command Validation - No Verification Abstract

The purpose of the "Stored Command Validation - No Verification" scenario is to describe the process by which a previously uplinked stored command for which no command verification is specified, is validated.

Figure 3.2.4.10-1 is the event trace diagram which corresponds to this scenario.

3.2.4.10.2 Stored Command Validation - No Verification Summary Information

Interfaces:

FOS User Interface

Telemetry Subsystem

Data Management Subsystem

Stimulus:

A previously uplinked stored command, for which no command verification is specified, is forwarded by FOS User Interface Subsystem.

Desired Response:

The stored command is successfully validated, but not added to the queue of commands to be verified.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

None.

3.2.4.10.3 Scenario Description

Note: this scenario is specific to the AM-1 mission.

FoGnCmdFuiIF provides to FcCdCmdController a previously uplinked, stored command in mnemonic format.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController queries the FcCdCommandDatabase for the requested command and gathers information about the command. The presence of the command within the database confirms the most basic syntax check: that the command mnemonic does in fact exist for the spacecraft being commanded.

FcCdCmdController then uses the information from the database query to check if verification is required for the command. No verification is required, so an event message is issued via FdEvEventLogger to that effect, and FcCdCmdController invokes the PutResponse function of the FoGnCmdFuiProxy to return an acknowledgment to user interface.

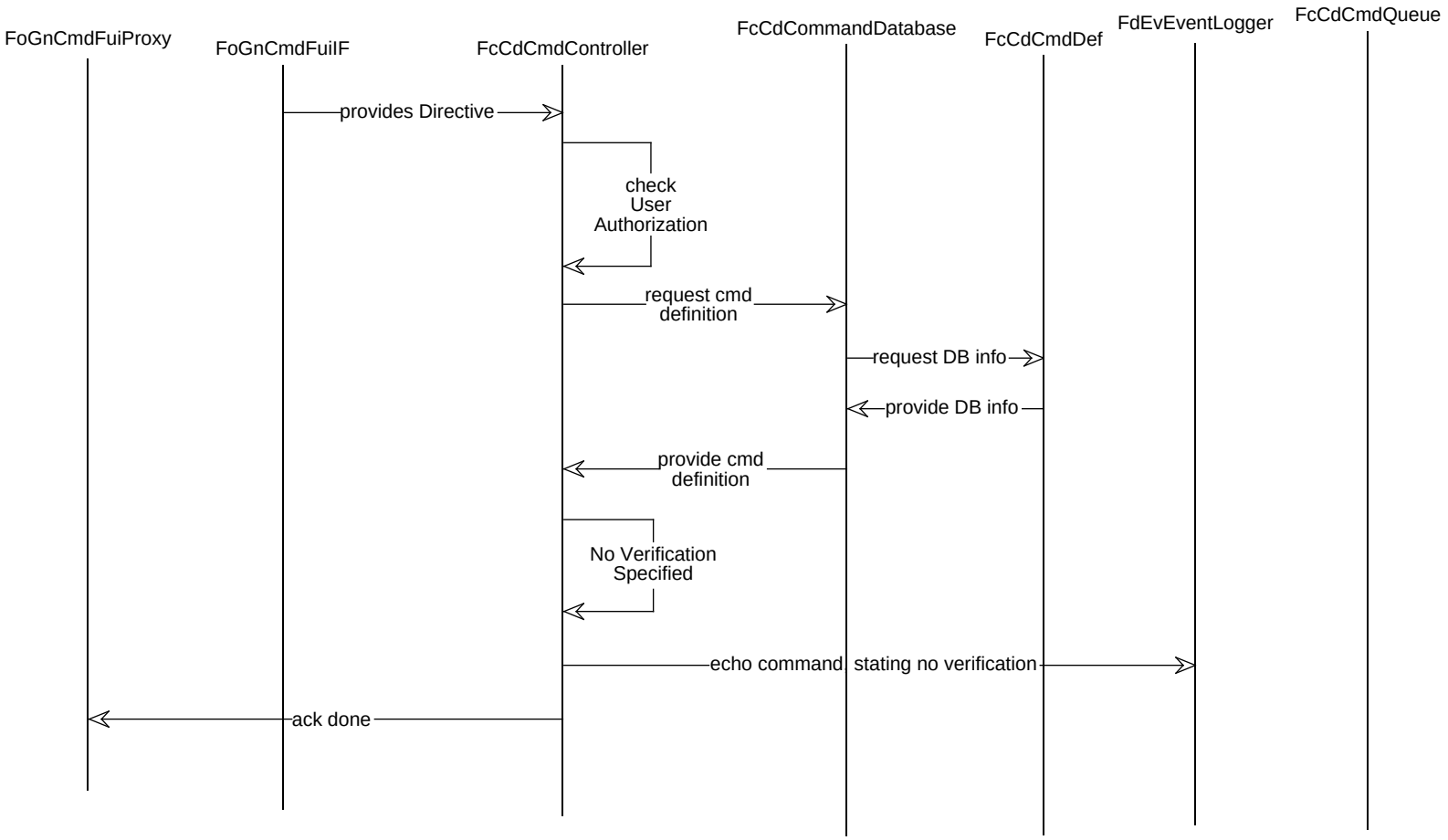

Figure 3.2.4.10-1. Stored Command Validation: No Verification required

3.2.4.11 Write Configuration Snapshot Request Scenario

3.2.4.11.1 Write Configuration Snapshot Request Abstract

The purpose of the "Write Configuration Snapshot Request" scenario is to describe the process by which the current state of a FormatCommand process is stored.

Figure 3.2.4.11-1 is the event trace diagram which corresponds to this scenario.

3.2.4.11.2 Write Configuration Snapshot Request Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A "Configuration Snapshot Request" is forwarded by the Resource Manager Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

Database must be identified and loaded.

Post-Conditions:

The state information is stored in the specified file.

3.2.4.11.3 Scenario Description

A Resource Managment "Configuration Snapshot Request" request is accepted by the Command Controller. The snapshot file specified in the request is accessed via FoDsFile, and the current state information is stored into the file. A successful status is returned to the Resource Manager.

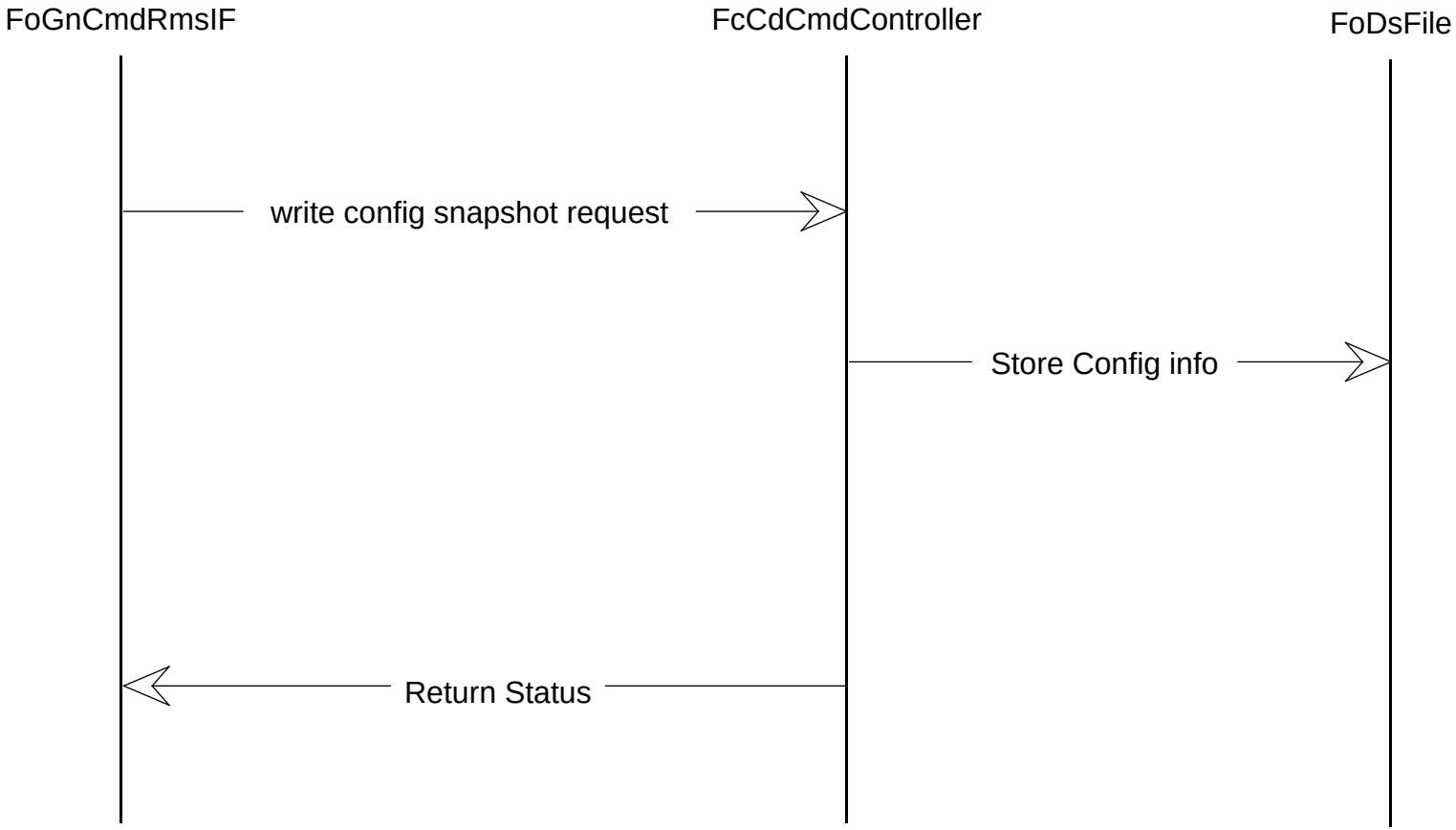

Figure 3.2.4.11-1. Write Configuration Snapshot request

3.2.4.12 Read Configuration Snapshot Request Scenario

3.2.4.12.1 Read Configuration Snapshot Request Abstract

The purpose of the "Read Configuration Snapshot Request" scenario is to describe the process by which the FormatCommand process is restored to a predetermined state.

Figure 3.2.4.12-1 is the event trace diagram which corresponds to this scenario.

3.2.4.12.2 Read Configuration Snapshot Request Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A request to "Read Configuration Snapshot" is forwarded by the Resource Manager Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.12.3 Scenario Description

A Resource Management "Read Configuration Snapshot" request is accepted by the Command Controller. The snapshot file specified in the request is accessed via FoDsFile, and the information retrieved is used to update the state of the FormatCommand process. A successful status is returned to the Resource Manager.

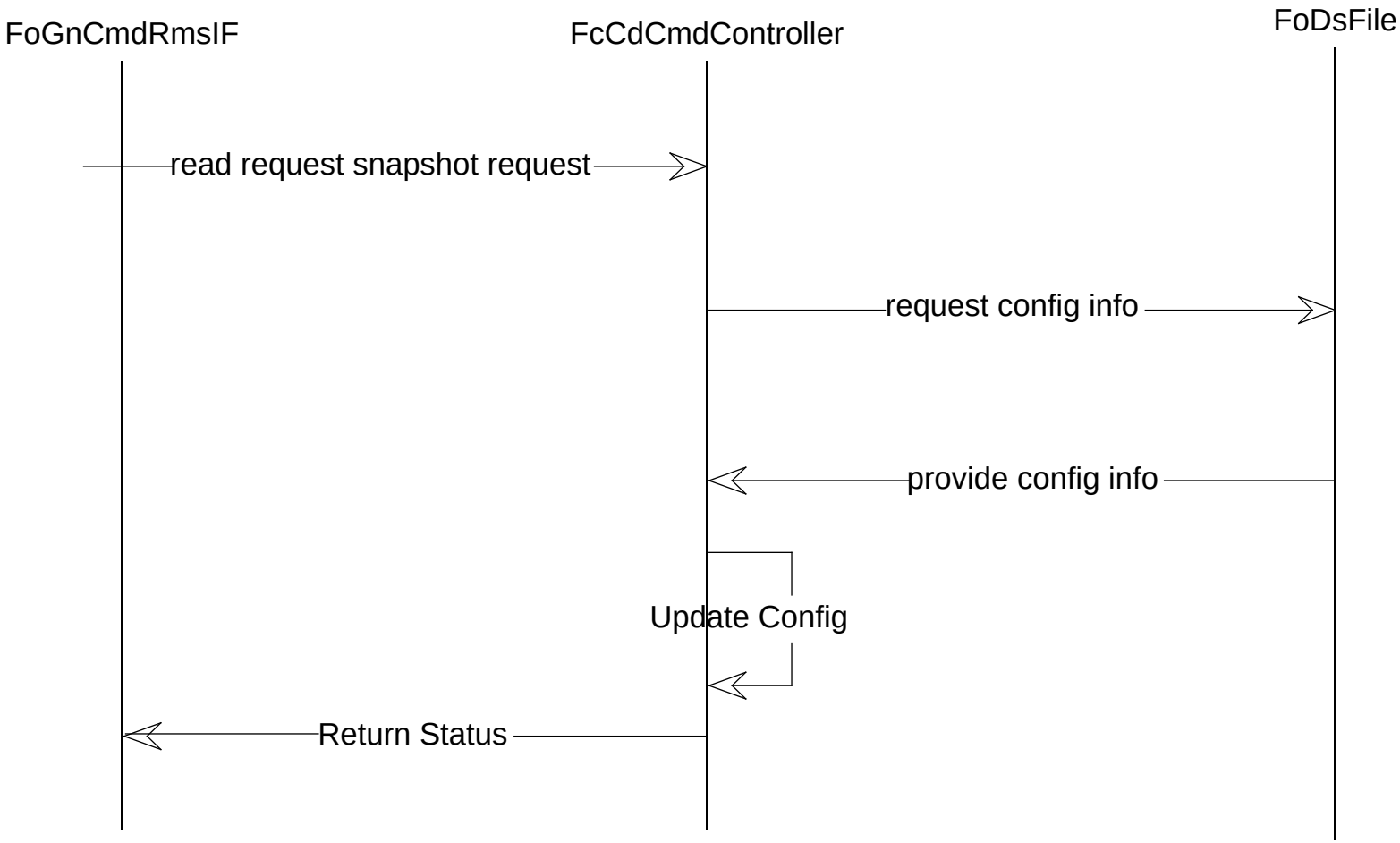

Figure 3.2.4.12-1. Read Configuration Snapshot request

3.2.4.13 Load Command Validation: Successful Scenario

3.2.4.13.1 Load Command Validation: Successful Abstract

The purpose of the "Load Command Validation: Successful" scenario is to describe the process by which a load file is processed for uplinking.

Figure 3.2.4.13-1 is the event trace diagram which corresponds to this scenario.

3.2.4.13.2 Load Command Validation: Successful Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

FopCommand

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of successful load command validation.

Pre-Conditions:

Catalog Entry and data for the load exist.

Post-Conditions:

The contents of the load file have been forwarded to FopCommand.

3.2.4.13.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which performs the following:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header.

It ensures that all prior partitions for this load have been uplinked. In this scenario, either one or more prior partitions have not been uplinked or the current load/partition was uplinked previously. The FcCdLoadData then issues a prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadData returns a status indicating waiting for user's response to FcCdLoadCmd which in turn returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPartitionRsp operation. The user's response is "allow" and validation processing continues. The ProcessPartitionRsp operation log event via FdEvEventLogger and performs the following:

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). (The CheckPrereq operation performs this function.) The prerequisite check is positive if the telemetry points are active (recently updated), and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPrereqRsp operation. The user's response is to allow; FcCdLoadCmd sends an event message via FdEvEventLogger and validation processing continues.

FcCdLoadCmd issues a critical prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessCriticalRsp operation. The user's response is to allow, the proper notifications are made via FdEvEventLogger and control returns to FcCdCmdController.

The load command is now successfully validated and the content is ready to be processed.

The load file is comprised of CCSDS packets. FcCdCmdController processes each packet in the file as follows:

It invokes the SendLoad operation of FcCdLoadCmd, which invokes the SendPacket operation of FcCdLoadData. This operation forwards one packet at a time to FopCommand via FoGnCmdFopProxy. FcCdCommandController waits for acknowledgment from FopCommand (via FoGnCmdFopIF) before proceeding to the next packet in the load file (a return status to FcCdCommandController indicates when the last packet of the load has been sent out).

Upon acknowledgment of the last packet, FcCdCommandController notifies FUI (via FjoGnCmdFuiIF) of completion of its request, and adds the FcCdLoadCmd to the queue of commands waiting verification.

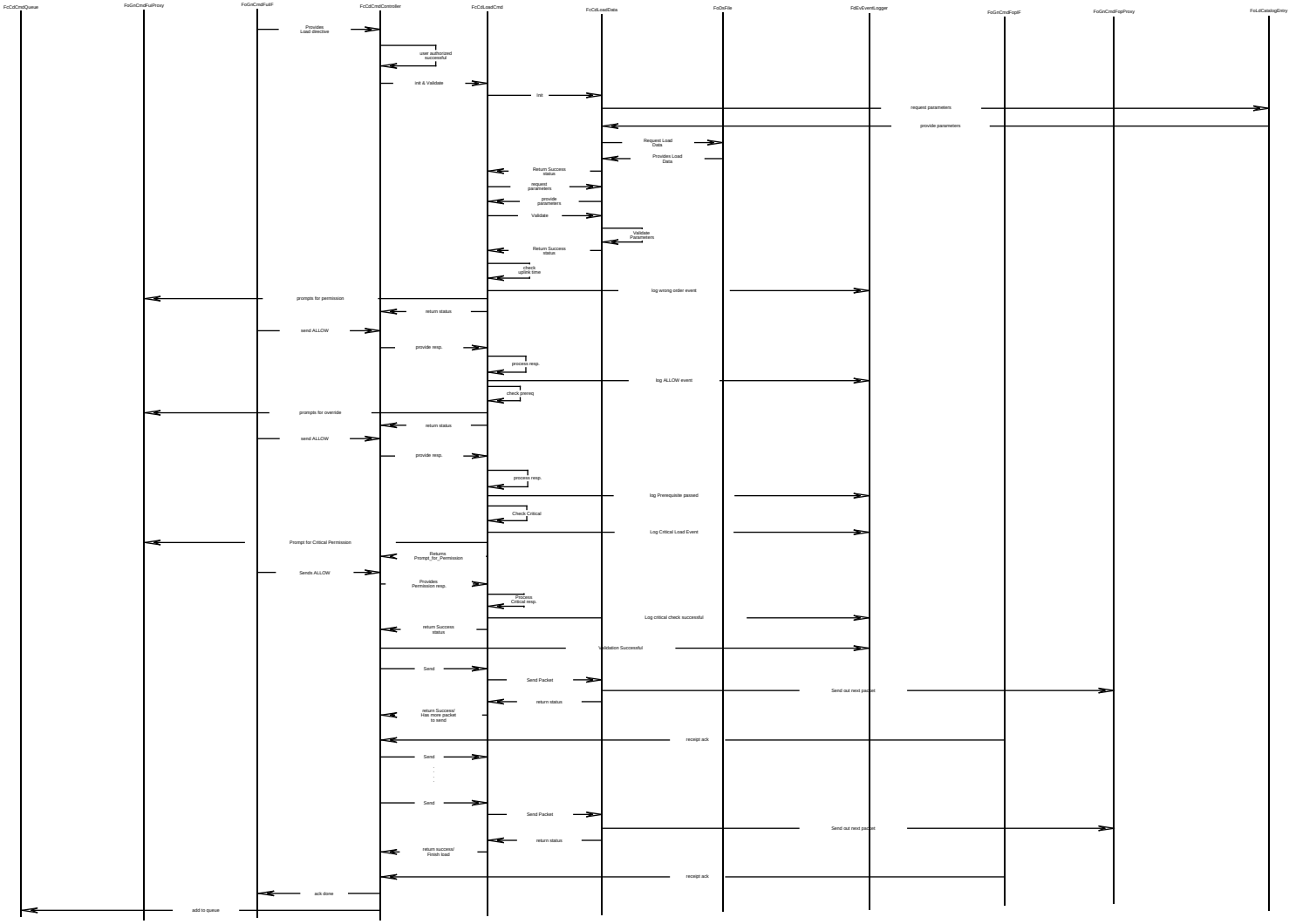

Figure 3.2.4.13-1. Load Command Validation: Successful Event Trace

3.2.4.14 Load Command Validation: Fail Due to Missing Load Scenario

3.2.4.14.1 Load Command Validation: Fail Due to Missing Load Abstract

The purpose of the "Load Command Validation: Fail Due to Missing Load" scenario is to describe the process by which the load is not uplinked due to missing load data.

Figure 3.2.4.14-1 is the event trace diagram which corresponds to this scenario.

3.2.4.14.2 Load Command Validation: Fail Due to Missing Load Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.14.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisitecheck (Pid, prerequisite type and ranges). In this scenario, there is no load data corresponding to the LoadId. The FcCdLoadData object logs an error message to FdEvEventLogger, provides error status to FUI and returns unsuccessful status to FcCdLoadCmd, which , in turn, returns unsuccessful status to FcCdCmdController.

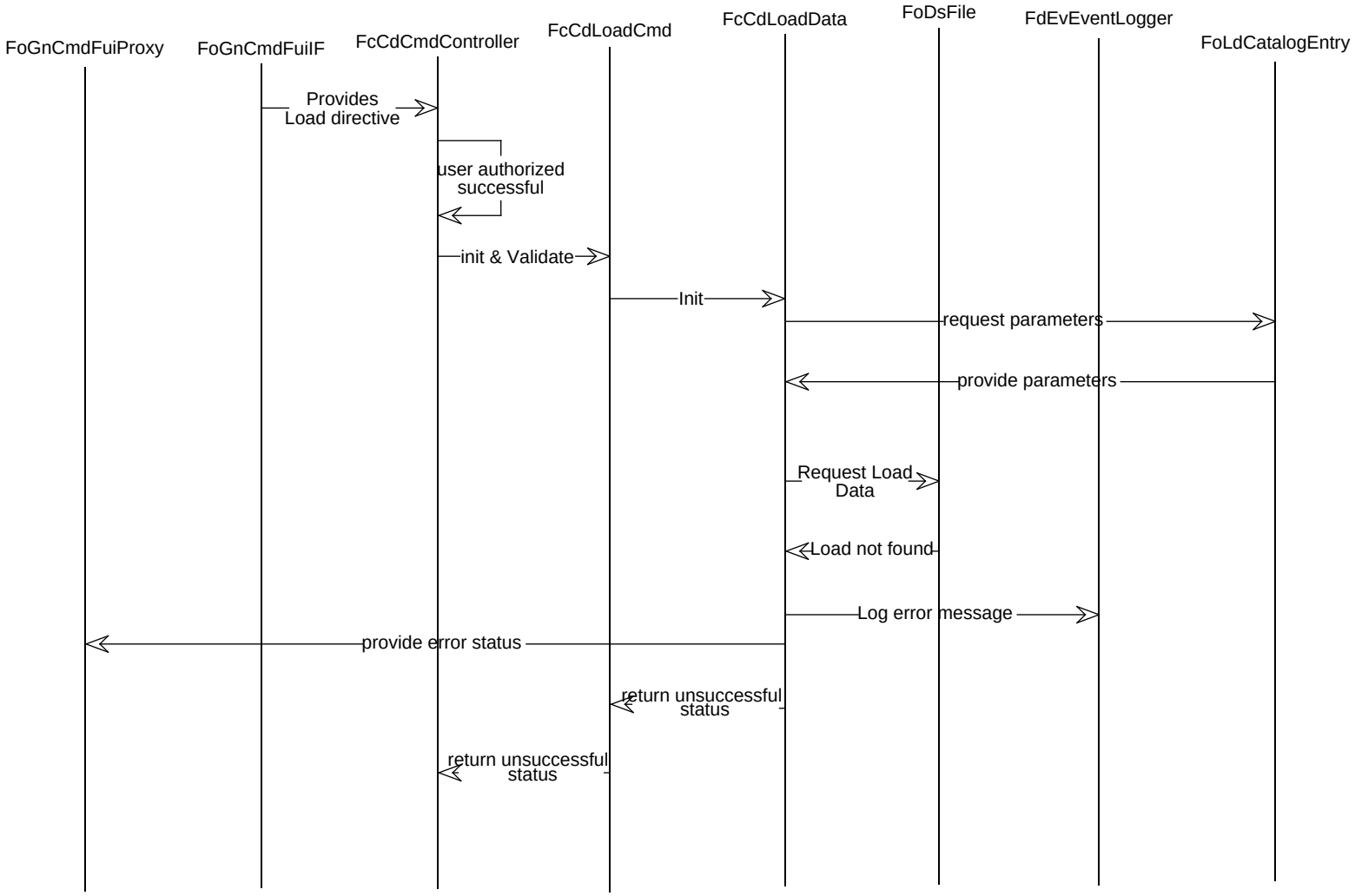

Figure 3.2.4.14-1. Load Command Validation: Unsuccessful due to missing load

3.2.4.15 Load Command Validation: Fail Due to Invalid Parameter Scenario

3.2.4.15.1 Load Command Validation: Fail Due to Invalid Parameter Abstract

The purpose of the "Load Command Validation: Fail Due to Invalid Parameter" scenario is to describe the process by which the load is not uplinked due to invalid load parameters.

Figure 3.2.4.15-1 is the event trace diagram which corresponds to this scenario.

3.2.4.15.2 Load Command Validation: Fail Due to Invalid Parameter Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.15.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which first performs the validation of load parameters:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header. In this scenario, either the spacecraftIds are not the same or the current time is outside the valid time window. The FcCdLoadData then logs an error message via FdEvEventLogger, sends an error status to FUI, and returns failure status to FcCdLoadCmd, which, in turn, return unsuccessful status to FcCdCmdController.

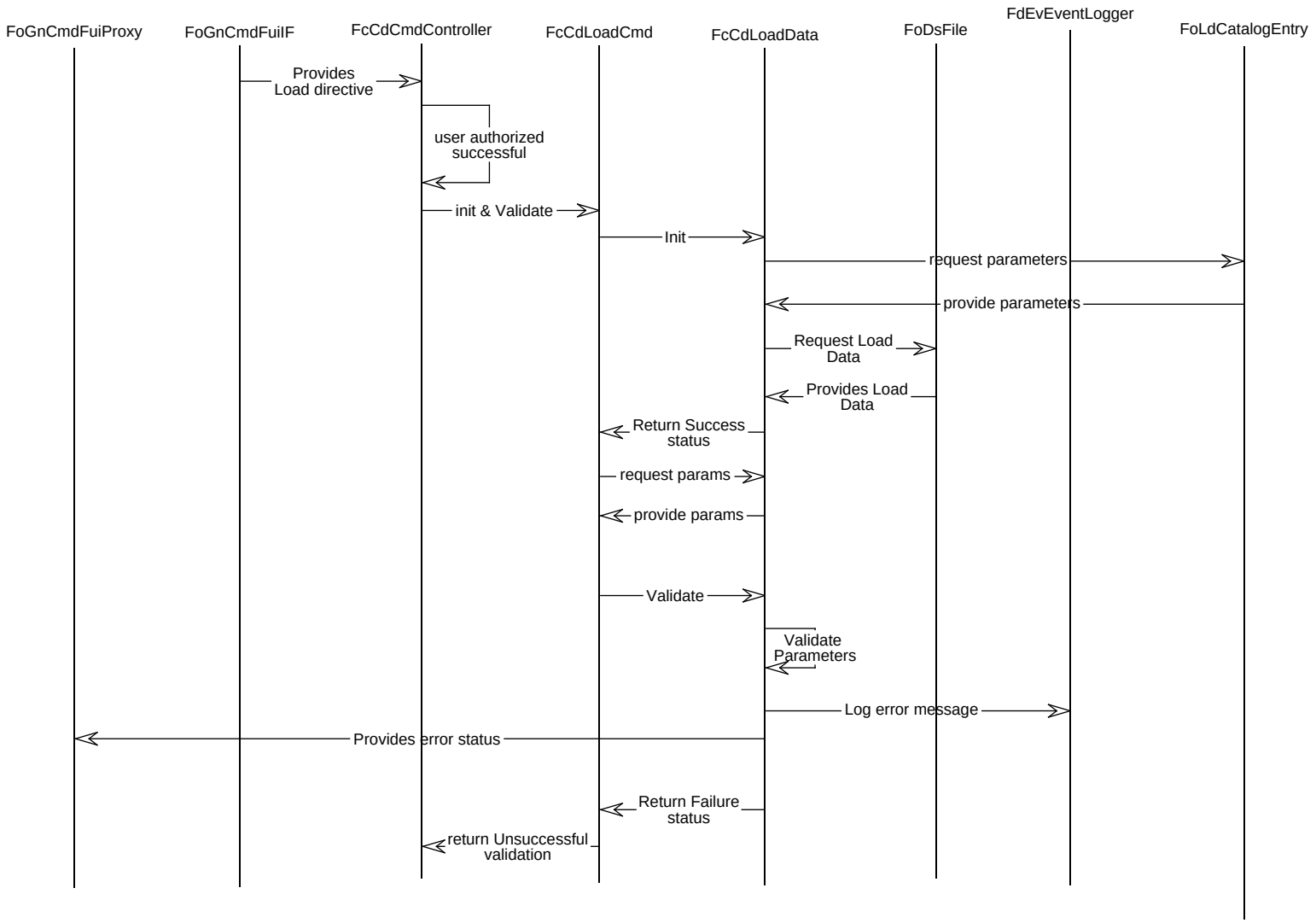

Figure 3.2.4.15-1. Load Command Validation: Unsuccessful due to Invalid Parameters

3.2.4.16 Load Command Validation: Fail Due to Cancel Out-of-Ordered Partition Scenario

3.2.4.16.1 Load Command Validation: Fail Due to Cancel Out-of-Ordered Partition Abstract

The purpose of the "Load Command Validation: Fail Due to Cancel Out-of-Ordered Partition" scenario is to describe the process by the load is not uplinked when user indicates Cancel to the Override-Out-of-Ordered prompt.

Figure 3.2.4.16-1 is the event trace diagram which corresponds to this scenario.

3.2.4.16.2 Load Command Validation: Fail Due to Cancel Out-of-Ordered Partition Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

One or more previous partitions have not been uplinked or the current load/partition was already uplinked.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.16.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which performs the following:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header.

It ensures that all prior partitions for this load have been uplinked. In this scenario, either one or more prior partitions have not been uplinked or the current load/partition was uplinked previously. The FcCdLoadData then issues a prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadData returns a status to FcCdLoadCmd, indicating waiting for user's response, which in turn returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its Process PartitionRsp operation. In this scenario, the user's response is "cancel". The FcCdLoadCmd object log cancel message to FdEvEventLogger, sends an ack to FUI and returns control to FcCdCmdController.

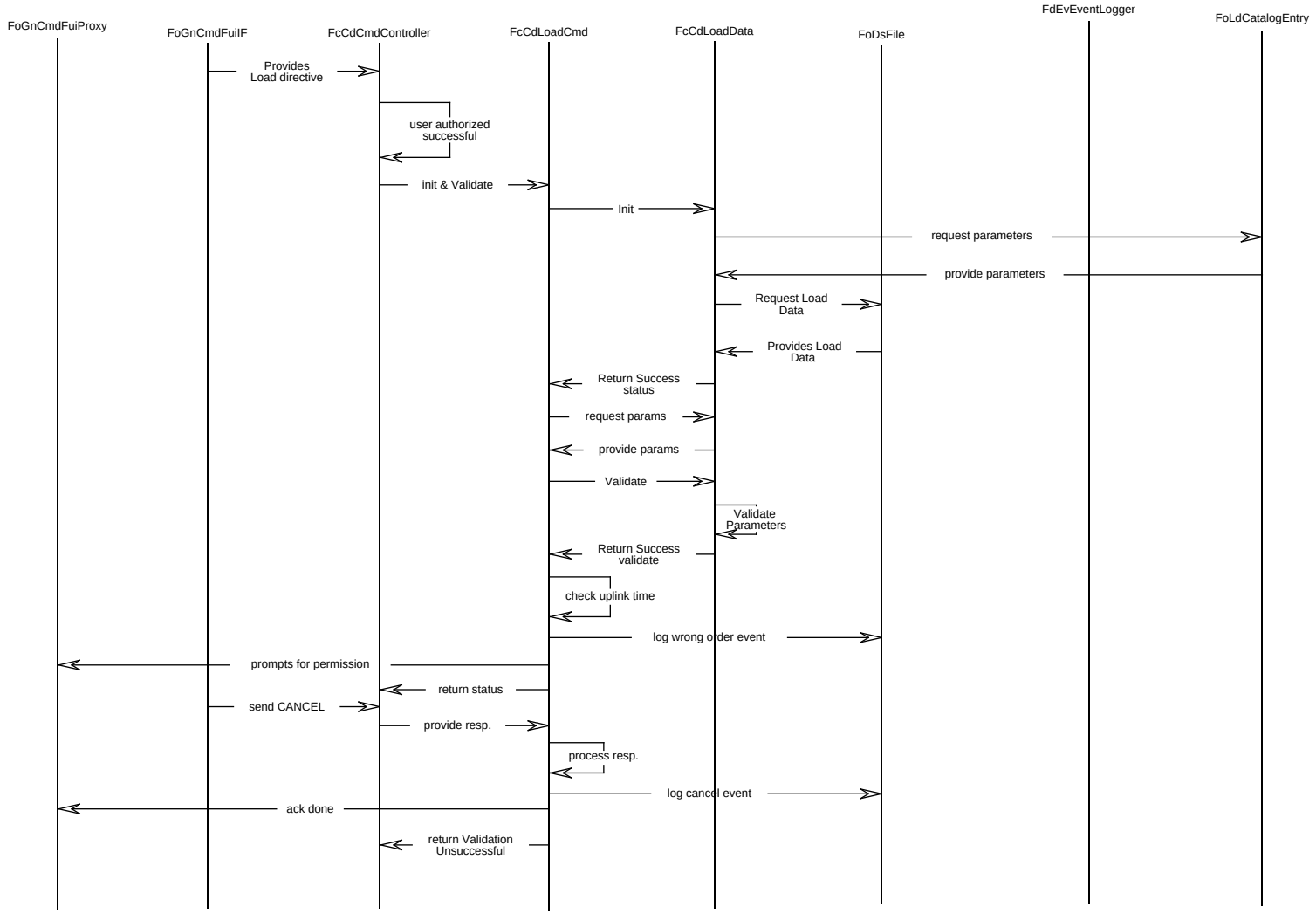

Figure 3.2.4.16-1. Load Command Validation: Unsuccessful due to canceling out-of-ordered partition

3.2.4.17 Load Command Validation: Fail Due to Cancel Prerequisite Override Scenario

3.2.4.17.1 Load Command Validation: Fail Due to Cancel Prerequisite Override Abstract

The purpose of the "Load Command Validation: Fail Due to Cancel Prerequisite Override" scenario is to describe the process by which the load is not uplinked when the user indicates cancel to the prerequisite override prompt.

Figure 3.2.4.17-1 is the event trace diagram which corresponds to this scenario.

3.2.4.17.2 Load Command Validation: Fail Due to Cancel Prerequisite Override Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.17.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which performs the following:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header.

It ensures that all prior partitions for this load have been uplinked. In this scenario, either one or more prior partitions have not been uplinked or the current load/partition was uplinked previously. The FcCdLoadData then issues a prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadData returns a status indicating waiting for user's response to FcCdLoadCmd which in turn returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPartitionRsp operation. The user's response is "allow" and validation processing continues. The ProcessPartitionRsp operation log event via FdEvEventLogger and performs the following:

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). (The CheckPrereq operation performs this function.) The prerequisite check is positive if the telemetry points are active (recently updated), and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the USER to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessPrereqRsp operation. The user's response is to cancel. The FcCdLoadCmd logs a message via FdEvEventLogger, sends an ack to FUI and returns unsuccessful status to FcCdCmdController.

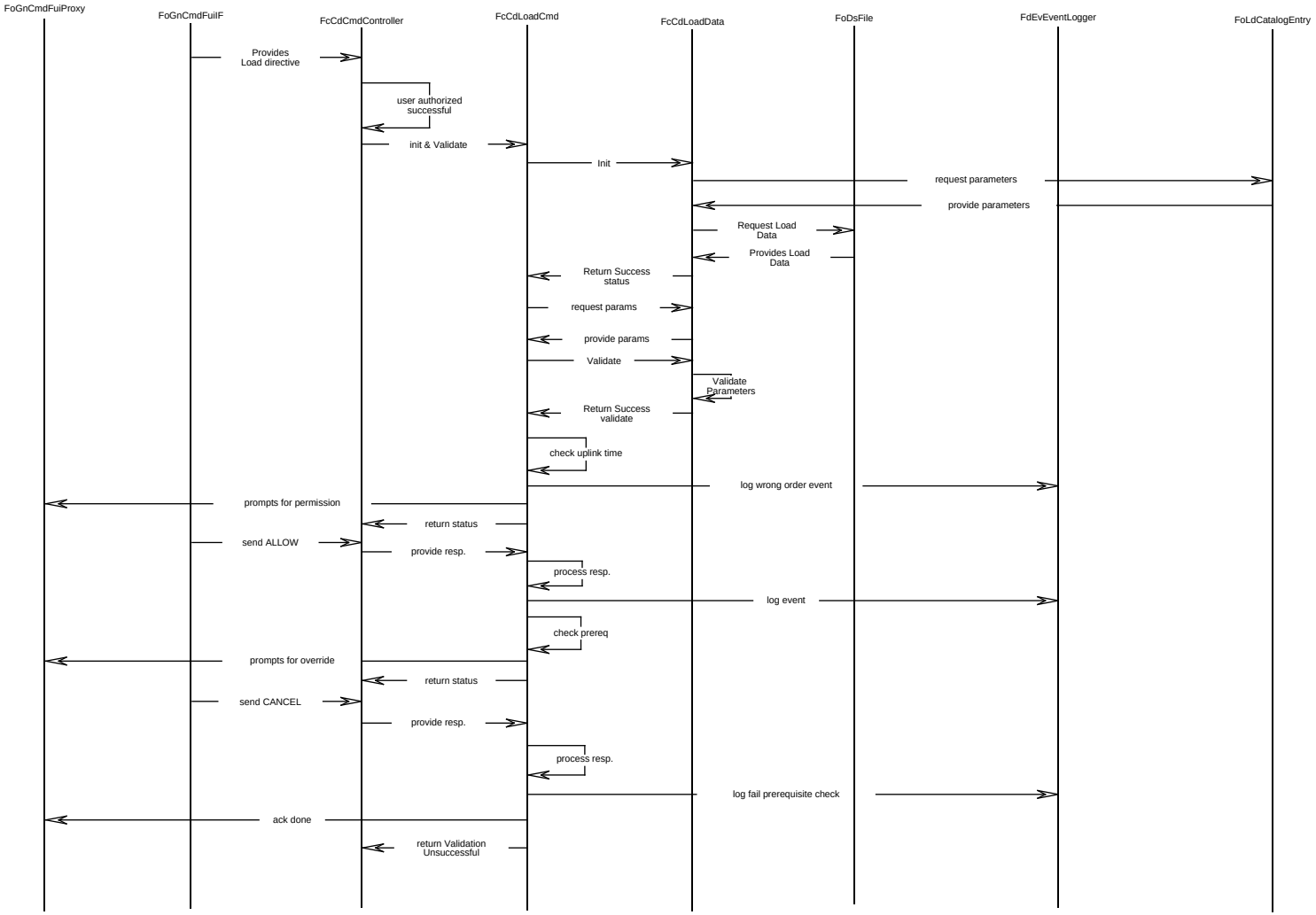

Figure 3.2.4.17-1. Load Command Validation: Unsuccessful due to no prerequisite override

3.2.4.18 Load Command Validation: Fail Due to Cancel Critical Scenario

3.2.4.18.1 Load Command Validation: Fail Due to Cancel Critical Abstract

The purpose of the "Load Command Validation: Fail Due to Cancel Critical" scenario is to describe the process by which the load is not uplinked when the user indicates cancel to the critical prompt.

Figure 3.2.4.18-1 is the event trace diagram which corresponds to this scenario.

3.2.4.18.2 Load Command Validation: Fail Due to Cancel Critical Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A load directive is forwarded by FOS User Interface Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.18.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which performs the following:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header.

It ensures that all prior partitions for this load have been uplinked. In this scenario, either one or more prior partitions have not been uplinked or the current load/partition was uplinked previously. The FcCdLoadData then issues a prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadData returns a status indicating waiting for user's response to FcCdLoadCmd which in turn returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPartitionRsp operation. The user's response is "allow" and validation processing continues. The ProcessPartitionRsp operation log event via FdEvEventLogger and performs the following:

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). (The CheckPrereq operation performs this function.) The prerequisite check is positive if the telemetry points are active (recently updated), and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPrereqRsp operation. The user's response is to allow; FcCdLoadCmd sends an event message via FdEvEventLogger and validation processing continues.

FcCdLoadCmd issues a critical prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessCriticalRsp operation. The user's response is to cancel. The FcCdLoadCmd logs a message via FdEvEventLogger, sends an ack to FUI and returns unsuccessful status to FcCdCmdController.

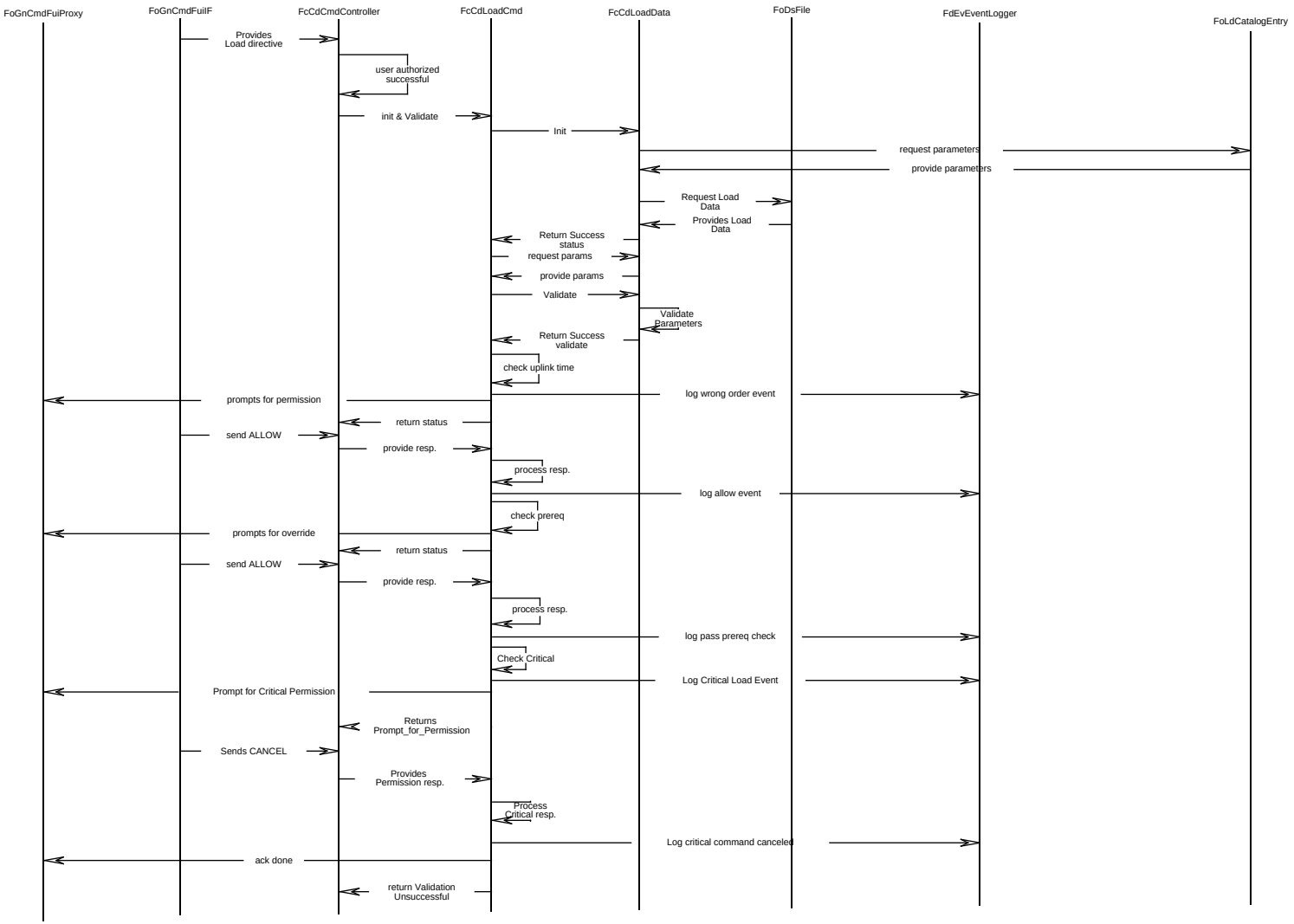

Figure 3.2.4.18-1. Load Command Validation: Unsuccessful due to canceling critical

3.2.4.19 Load Command Validation: Abort Scenario

3.2.4.19.1 Load Command Validation: Abort Abstract

The purpose of the "Load Command Validation: Abort" scenario is to describe the process by which a load file processing is stopped via an abort directive.

Figure 3.2.4.19-1 is the event trace diagram which corresponds to this scenario.

3.2.4.19.2 Load Command Validation: Abort Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

FopCommand

Stimulus:

A load command is forwarded by FOS User Interface Subsystem.

Desired Response:

FOS User Interface receives the status of successful load command validation/generation.

Pre-Conditions:

None.

Post-Conditions:

The contents of the load file have been forwarded to FopCommand.

3.2.4.19.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time directive. The directive in the scenario is a load directive to process a load with critical commands.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

FcCdCmdController then creates an FcCdLoadCmd object. The FcCdLoadCmd object, in turn, instantiates a FcCdLoadData object and initializes it with the LoadId. It then invokes the Init operation of FcCdLoadData. In this operation, the load catalog entry and data (i.e., the load packets) are read in. The load catalog entry contains : critical flag, destination, spacecraft ID, Time window, CRC, number of packets for the load (or partition of load), information for telemetry verification (telemetry PID and timeout/wait interval), and information for prerequisite check (P(Pid, prerequisite type and ranges). The FcCdLoadCmd object then invokes the GetParameters operation of the FcCdLoadData object to obtain information for telemetry verification.

FcCdCmdController then invokes the Validate operation of the FcCdLoadCmd, which performs the following:

It ensures that the load file is intended for the spacecraft being commanded by the current process.

It ensures that the current time is within the valid window period of time specified in the header.

It ensures that all prior partitions for this load have been uplinked. In this scenario, either one or more prior partitions have not been uplinked or the current load/partition was uplinked previously. The FcCdLoadData then issues a prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadData returns a status indicating waiting for user's response to FcCdLoadCmd which in turn returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPartitionRsp operation. The user's response is "allow" and validation processing continues. The ProcessPartitionRsp operation log event via FdEvEventLogger and performs the following:

It compares one (1) to four (4) database defined state(s) of the prerequisite telemetry point(s) against their current state(s). (The CheckPrereq operation performs this function.) The prerequisite check is positive if the telemetry points are active (recently updated), and match the prerequisite states. In this scenario, however, one or more telemetry point(s) is either inactive or does not meet the prerequisite states, thus yielding a negative prerequisite check. It then issues a prerequisite override prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController.

FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessPrereqRsp operation. The user's response is to allow; FcCdLoadCmd sends an event message via FdEvEventLogger and validation processing continues.

FcCdLoadCmd issues a critical prompt to the user to respond allow or cancel. As this is an asynchronous communication, FcCdLoadCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdLoadCmd by invoking its ProcessCriticalRsp operation. The user's response is to allow, the proper notifications are made via FdEvEventLogger and control returns to FcCdCmdController.

The load command is now successfully validated and the content is ready to be processed.

The load file is comprised of CCSDS packets. FcCdCmdController processes each packet in the file as follows:

It invokes the SendLoad operation of FcCdLoadCmd, which invokes the SendPacket operation of FcCdLoadData. This operation forwards one packet at a time to FopCommand via FoGnCmdFopProxy. FcCdCommandController waits for acknowledgment from FopCommand (via FoGnCmdFopIF) before proceeding to the next packet in the load file (a return status to FcCdCommandController indicates when the last packet of the load has been sent out).

In this scenario, after several packets are forwarded to FopCommand, an abort directive is received. The FcCdCmdController destructs the FcCdLoadCmd object, which in turn destructs the FcCdLoadData object. The FcCdCmdController then sends an ack to FUI.

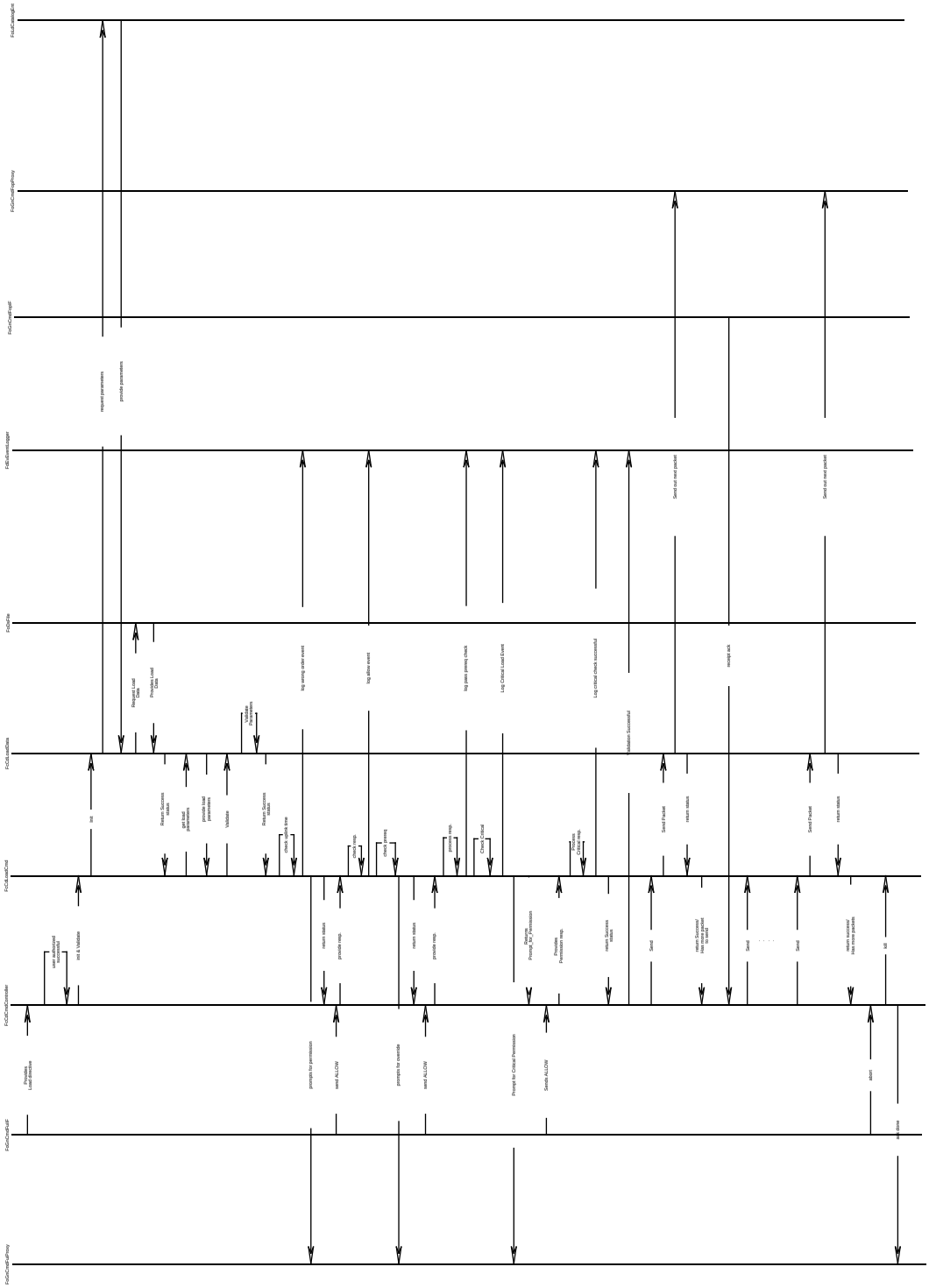

Figure 3.2.4.19-1. Load Command: Abort Load

3.2.4.20 Real-Time Command Verification: Success Scenario

3.2.4.20.1 Real-Time Command Verification: Success Abstract

The purpose of the "Real-Time Command Verification: Success" scenario is to describe the process by which real time commands are telemetry verified.

Figure 3.2.4.20-1 is the event trace diagram which corresponds to this scenario.

3.2.4.20.2 Real-Time Command Verification: Success Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

Telemetry Subsystem

FopCommand

Stimulus:

A command receipt is received from FopCommand.

Desired Response:

User Interface is notified of telemetry verification of the command.

Pre-Conditions:

The command queue contains at least one command: the command corresponding to the command receipt mentioned above.

Post-Conditions:

The command corresponding to the command receipt has been removed from the command queue.

3.2.4.20.3 Scenario Description

FcCdCommandController receives a command receipt from FopCommand via FoCmCCSDSFopProxy, and FUI is notified of the command's upgraded status via FoGnCmdFuiIF. FcCdCmdQueue is requested to begin verification of the command if there is a verification parameter. The command (FcCdCmd) is found in the queue, and the parameter service list is updated to reflect the new telemetry parameter required for verification of this command through FoGnCmdTlmProxy.

As telemetry is received, FcCdCommandController is sent a list of updated telemetry parameters via the TlmParamerterServerProxy. This list is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed. For each PID, all outstanding commands (FcCdCmd objects) which need that PID for verification are checked. The telemetry value supplied in this list is not within the range required to verify the command, so the command remains unverified at this time. This sequence may be repeated several times before verification takes place.

FcCdCommandController is eventually sent a list of updated telemetry parameters one of which will verify the traced command via the TlmParamerterServerProxy which is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed and for each PID, all outstanding commands (FcCdCmd objects) which need that PID for verification are checked. The telemetry

value supplied in this list however, is within the range required to verify the command, and the command is thus telemetry verified.

FUI is notified of the telemetry verification status via FoGnCmdFuiIF, an event message to that effect is logged via FdEvEventLogger, and the command is removed from FcCdCmdQueue.

After all of the telemetry PIDs have been checked, the PID list is updated to reflect only those PIDs needed for the current (i.e., smaller) list of outstanding commands in FcCdCmdQueue which still need to be telemetry verified. This revised PID list is then sent to TlmParameterServerProxy.

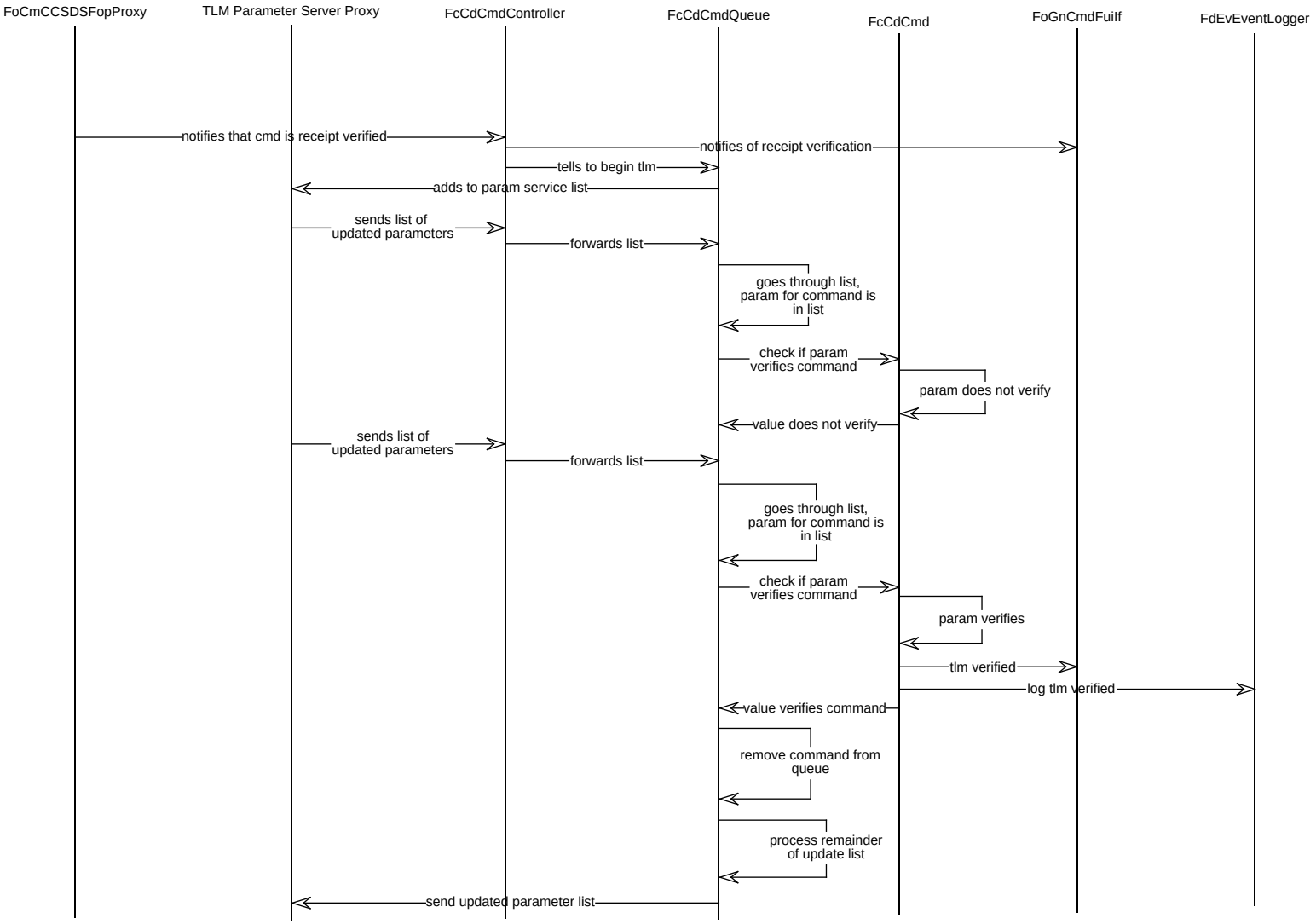

Figure 3.2.4.20-1. Real-Time Command Verification: Successful Event Trace

3.2.4.21 Real-Time Command Verification: Failure Due to Timeout Scenario

3.2.4.21.1 Real-Time Command Verification: Failure Due to Timeout Abstract

The purpose of the "Real-Time Command Verification: Failure Due to Timeout" scenario is to describe the process by which real-time commands which have not telemetry verified in the database prescribed time limit are taken off the queue and proper notification is accomplished

Figure 3.2.4.21-1 is the event trace diagram which corresponds to this scenario.

3.2.4.21.2 Real-Time Command Verification: Failure Due to Timeout Summary Information

Interfaces:

TLM, FUI, DMS, FopCommand

Stimulus:

It is found that the command has not been verified in the allowed time period.

Desired Response:

The command is removed from the queue and appropriate notification is done..

Pre-Conditions:

None.

Post-Conditions:

The command is no longer on the queue.

3.2.4.21.3 Scenario Description

FcCdCommandController receives a command receipt from FopCommand via FoCmCCSDSFopProxy, and FUI is notified of the command's status via FoGnCmdFuiIF. FcCdCmdQueue is requested to begin verification of the command if there is a verification parameter. The command (FcCdCmd) is found in the queue, and the parameter service list is updated to reflect the new telemetry parameter required for verification of this command through FoGnCmdTlmProxy.

As telemetry is received, FcCdCommandController is sent a list of updated telemetry parameters via the TlmParamerterServerProxy. This list is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed. For each PID, all outstanding commands (FcCdCmd objects) which need that PID for verification are checked. The telemetry value supplied in this list is not within the range required to verify the command, so the command remains unverified at this time. This sequence is repeated several times but none of the supplied updates of the parameter verifies the command. Meanwhile, at regular intervals, triggered by a timer, FcCdCmdController calls FcCdCmdQueue::CheckTimes() which then polls each command on the queue to see if it has timed out. Eventually, after the traced command has been on the queue for the maximum time, CheckTimes() is told by the traced command (FcCdRtCmd) that it has timed out. The command itself logs this event and notifies Fui, and the queue removes the command, and updates the parameter service list.

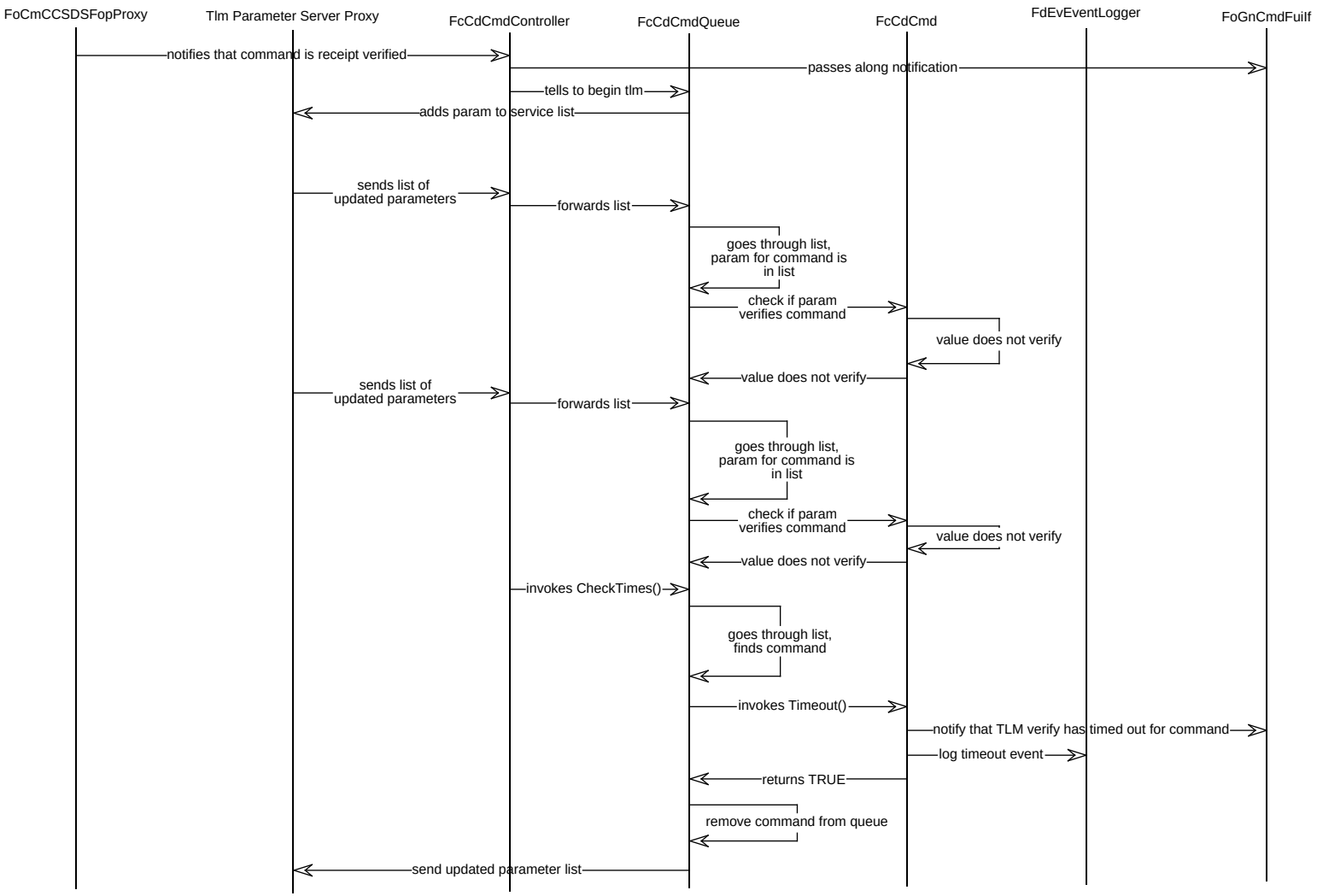

Figure 3.2.4.21-1. Real-Time Command Verification: Fail due to time out

3.2.4.22 Real-Time Load Verification: Success Scenario

3.2.4.22.1 Real-Time Load Verification: Success Abstract

The purpose of the "Real-Time Load Verification: Success" scenario is to describe the process by which real time loads are telemetry verified.

Figure 3.2.4.22-1 is the event trace diagram which corresponds to this scenario.

3.2.4.22.2 Real-Time Load Verification: Success Summary Information

Interfaces:

FOS User Interface

Data Management Subsystem

Telemetry Subsystem

FopCommand

Stimulus:

A command receipt is received from FopCommand.

Desired Response:

User Interface is notified of telemetry verification of the load.

Pre-Conditions:

The command queue contains at least one command; the command corresponding to the load receipt mentioned above.

Post-Conditions:

The command corresponding to the load receipt has been removed from the command queue.

3.2.4.22.3 Scenario Description

FcCdCommandController receives a command receipt that corresponds to the last packet in the load, from FopCommand (via FoCmCCSDSFopProxy). FUI is notified of the load's upgraded (uplink verified) status via FoGnCmdFuiIF. FcCdCmdQueue is requested to begin verification of the load. The command (FcCdCmd) corresponding to the load is found in the queue, and the parameter service list is updated to reflect the new telemetry parameters (i.e., the CRC) required for verification of the load through FoGnCmdTlmProxy.

As telemetry is received, FcCdCommandController is sent a list of updated telemetry parameters via the TlmParamerterServerProxy. This list is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed. For each PID, all outstanding commands (FcCdCmd objects) which need that PID for verification are checked. The telemeter CRC value supplied in this list does not confirm the load verification the command, so the load remains unverified at this time.

FcCdCommandController is then sent a second list of updated telemetry parameters via the TlmParamerterServerProxy which is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed and for each PID, all outstanding commands (FcCdCmd objects) which need that PID for verification are checked. The telemeter CRC value supplied in this list this time matches the CRC retained from the load header (as myCRCRWCString), and the load is thus telemetry verified.

FUI is notified of the telemetry verification status via FoGnCmdFuiIF, an event message to that effect is logged via FdEvEventLogger, and the load command is removed from FcCdCmdQueue.

After all of the telemetry PIDs have been checked, the PID list is updated to reflect only those PIDs needed for the current (i.e., smaller) list of outstanding commands in FcCdCmdQueue which still need to be telemetry verified. This revised PID list is then sent to TlmParameterServerProxy.

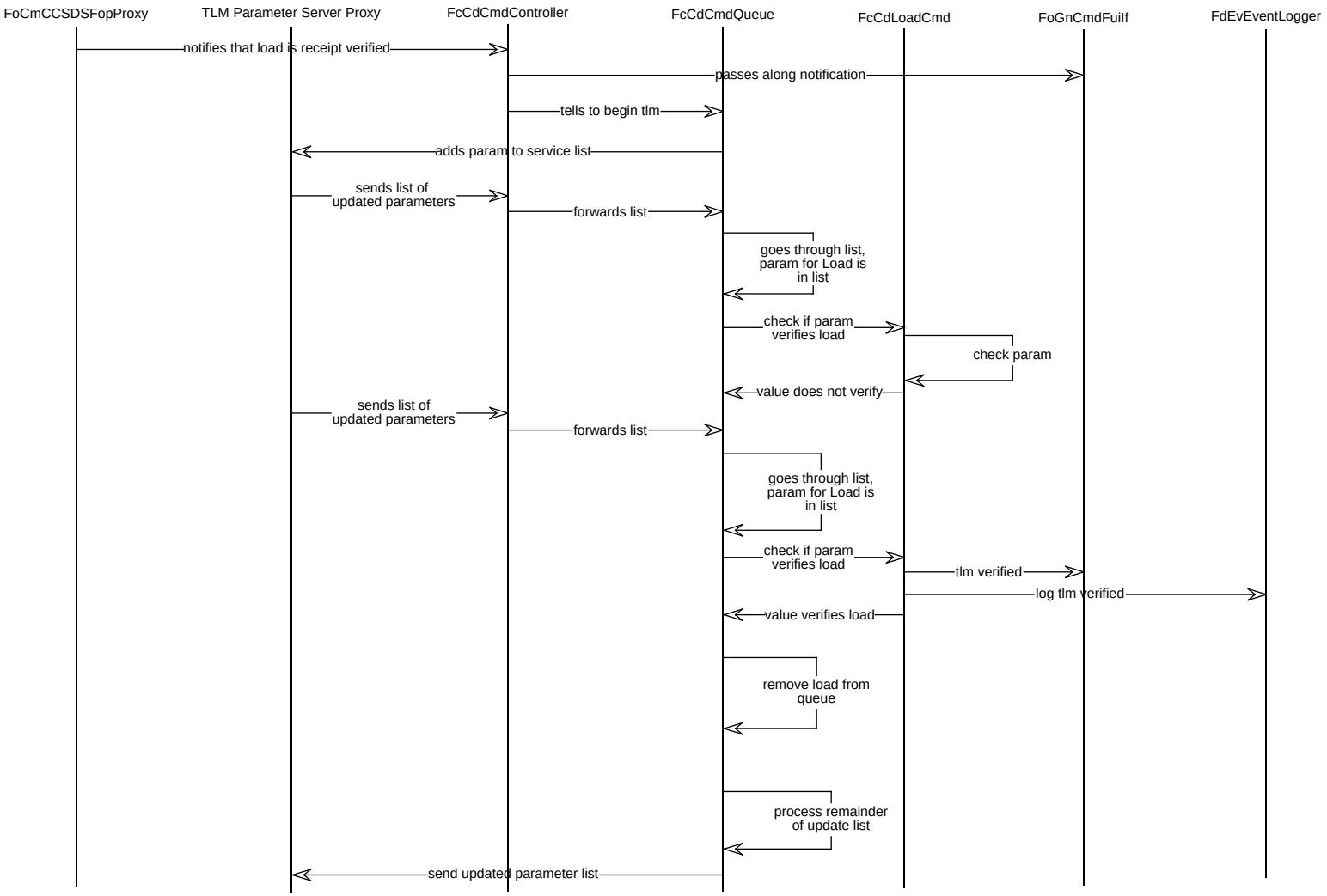

Figure 3.2.4.22-1. Real-Time Load Verification: Successful Event Trace

3.2.4.23 Real-Time Load Verification: Failure Due to Timeout Scenario

3.2.4.23.1 Real-Time Load Verification: Failure Due to Timeout Abstract

The purpose of the "Real-Time Load Verification: Failure Due to Timeout" scenario is to describe the process by which real-time load commands which have not telemetry verified in the database prescribed time limit are taken off the queue and proper notification is accomplished

Figure 3.2.4.23-1 is the event trace diagram which corresponds to this scenario.

3.2.4.23.2 Real-Time Load Verification: Failure Due to Timeout Summary Information

Interfaces:

TLM, DMS, FUI

Stimulus:

It is found that the load has not been verified in the allowed time period.

Desired Response:

The load command is removed from the queue and notification is done.

Pre-Conditions:

None.

Post-Conditions:

The load command is no longer on the queue.

3.2.4.23.3 Scenario Description

FcCdCommandController receives a load command receipt from FopCommand via FoCmCCSDSFopProxy, and FUI is notified of the load command's upgraded status via FoGnCmdFuiIF. FcCdCmdQueue is requested to begin verification of the load command. The load command (FcCdLoadCmd) is found in the queue, and the parameter service list is updated to reflect the new telemetry parameter required for verification of this command through FoGnCmdTlmProxy.

As telemetry is received, FcCdCommandController is sent a list of updated telemetry parameters via the TlmParamerterServerProxy. This list is forwarded to FcCdCmdQueue. The list of telemetry PIDs is traversed. For each PID, all outstanding commands, including loads, (FcCdCmd objects) which need that PID for verification are checked. The telemetry value supplied in this list is not within the range required to verify the load command, so the load command remains unverified at this time. This sequence is repeated several times but none of the supplied updates of the parameter verifies the load command.

Meanwhile, at regular intervals, triggered by a timer, FcCdCmdController calls FcCdCmdQueue::CheckTimes() which then polls each command on the queue to see if it has timed out. Eventually, after the traced load command has been on the queue for the maximum time, CheckTimes() is told by the traced load command (FcCdLoadCmd) that it has timed out. The load command itself logs this event and notifies Fui, and the queue removes the load command, and updates the parameter service list.

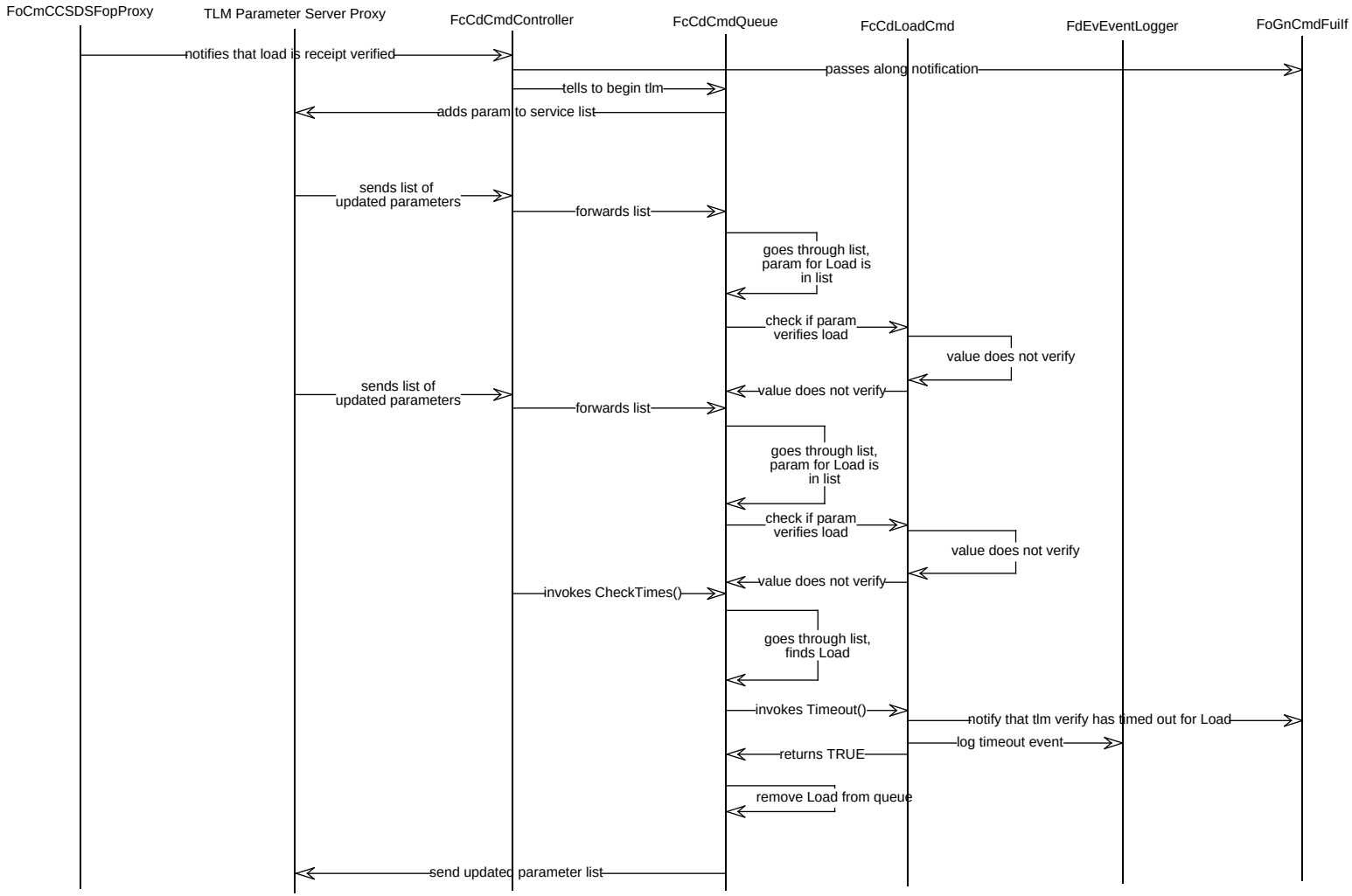

Figure 3.2.4.23-1. Real-Time Load Verification: Failure due to time out

3.2.4.24 Real-Time Dump Command Scenario

3.2.4.24.1 Real-Time Dump Command Abstract

The purpose of the "Real-Time Command Verification: Success" scenario is to describe the process by which real time commands are telemetry verified.

Figure 3.2.4.24-1 is the event trace diagram which corresponds to this scenario.

3.2.4.24.2 Real-Time Dump Command Summary Information

Interfaces:

TLM, DMS

Stimulus:

A dump command is received by FcCdCmdController.

Desired Response:

TLM is notified of the impending dump.

Pre-Conditions:

None.

Post-Conditions:

TLM has been warned.

3.2.4.24.3 Scenario Description

The dump notification scenario is identical to the realtime command validation success scenario, except:

When the command database is accessed by FcCdCmdController, the dump command flag in the database is set. After the command is validated, FcCdCmdController checks this flag, and since it is set, it sends a notification to TLM that a dump is imminent.

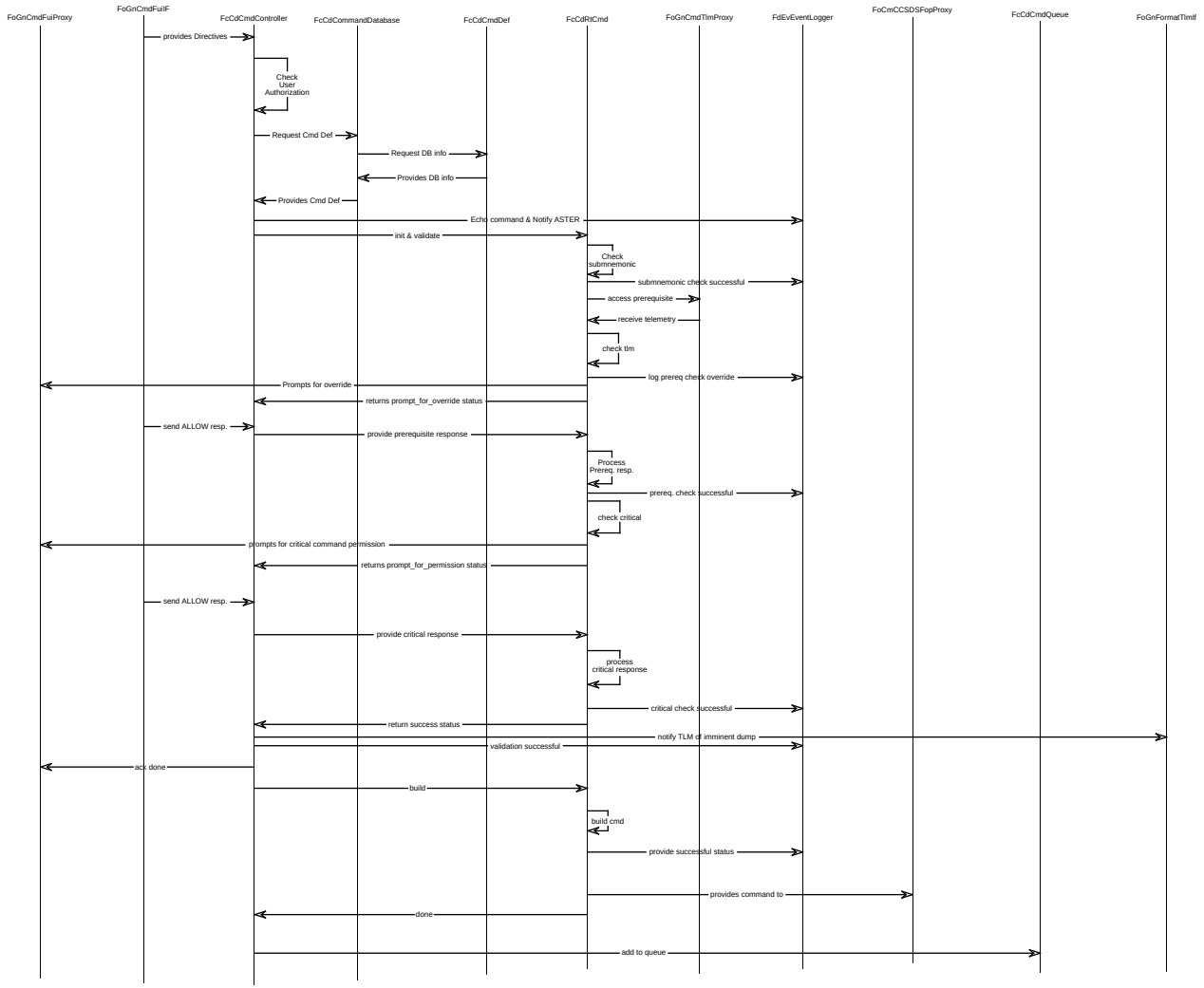

Figure 3.2.4.24-1. Real Time Dump

3.2.4.25 Hex Command Validation: Success Scenario

3.2.4.25.1 Hex Command Validation: Success Abstract

The purpose of the "Hex Command Validation: Success" scenario is to describe the process by which the Hex command is successfully validated.

Figure 3.2.4.25-1 is the event trace diagram which corresponds to this scenario.

3.2.4.25.2 Hex Command Validation: Success Summary Information

Interfaces:

FOS User Interface

FopCommand

Data Management Subsystem

Stimulus:

A hex formatted command is forwarded by FOS User Interface to the Command Subsystem.

Desired Response:

FOS User Interface receives the status of successful command validation/generation.

Pre-Conditions:

None.

Post-Conditions:

The command has been forwarded to FopCommand for eventual uplinking.

3.2.4.25.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time command, in hex format; i.e., already in 1553-b format.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

A FcCdHexCmd object is created, and its CheckCritical operation is invoked. A critical prompt is issued to the authorized user to respond allow or cancel. As this is an asynchronous communication, FcCdRtCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdRtCmd by invoking its ProcessCriticalRsp operation. The user's response is to allow, the proper notifications are made via FdEvEventLogger and control returns to FcCdCmdController.

FcCdCmdController invokes the SendCmd operation of FcCdHexCmd. The command is then forwarded to FopCommand via FoGnCmdFopProxy. Upon receipt of acknowledgment from FopCommand (via FoGGnCmdFopIF), FcCdCommandController notifies FUI (via FjoGnCmdFuiIF) of completion of its request, and adds the FcCdHexCmd to the queue of commands waiting uplink verification.

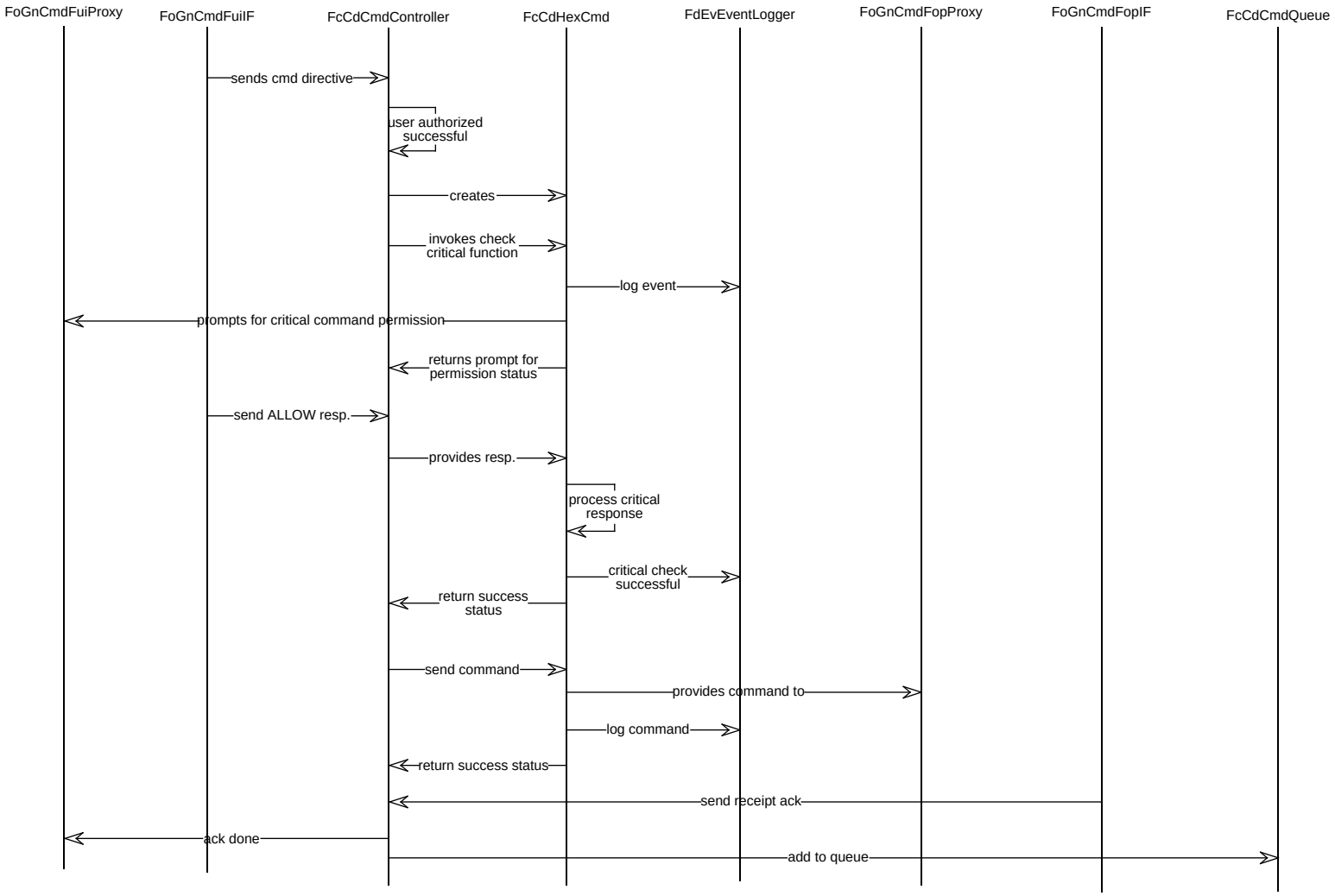

Figure 3.2.4.25-1. Hex Command Validation: Success Event Trace

3.2.4.26 Hex Command Validation: Fail Due to Cancel Critical Scenario

3.2.4.26.1 Hex Command Validation: Fail Due to Cancel Critical Abstract

The purpose of the "Hex Command Validation: Fail Due to Cancel Critical" scenario is to describe the process by a hex command is not uplinked when the user indicates cancel to the critical prompt.

Figure 3.2.4.26-1 is the event trace diagram which corresponds to this scenario.

3.2.4.26.2 Hex Command Validation: Fail Due to Cancel Critical Summary Information

Interfaces:

Resource Manager Interface

Data Management Subsystem

Stimulus:

A hex formatted command is forwarded by FOS User Interface to the Command Subsystem.

Desired Response:

The Resource Manager is notified of the completion.

Pre-Conditions:

None.

Post-Conditions:

The FormatCommand process is reconfigured.

3.2.4.26.3 Scenario Description

FoGnCmdFuiIF provides to FcCdCmdController a real-time command, in hex format; i.e., already in 1553-b format.

FcCdCmdController compares the ID and workstation of the issuer of the command against the ID of the currently authorized operator and workstation, myUserId and myWksId, to verify that, in fact, it is the operator with current command authorization who issued the command. The comparison shows that the IDs match.

A FcCdHexCmd object is created, and its CheckCritical operation is invoked. A critical prompt is issued to the authorized user to respond allow or cancel. As this is an asynchronous communication, FcCdHexCmd returns control to the FcCdCmdController, which resumes polling for all possible messages. FcCdCmdController then receives a FUI message indicating the user's response. It passes this response to FcCdHexCmd by invoking its ProcessCriticalRsp operation. The user's response is to cancel. The FcCdHexCmd log message via FdEvEventLogger , sends an ack to FUI and returns control to FcCdCmdController.

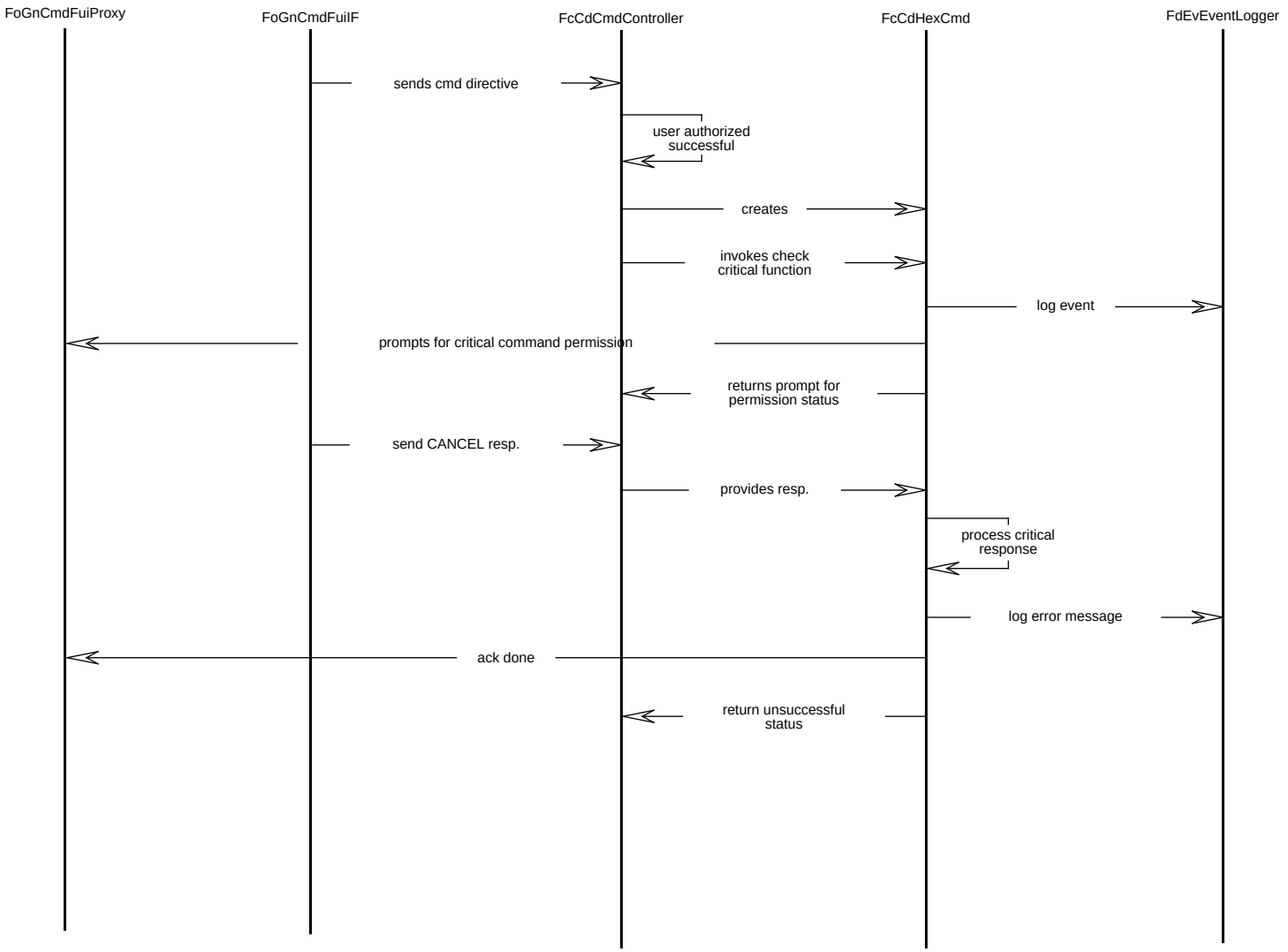

Figure 3.2.4.26-1. Hex Command Validation: Failure Event Trace

3.2.4.27 FcCdCmdController State Diagram

Once initialized, the controller object enters a superstate called "SuperWait" in the diagram. From this state, it can process messages from other subsystems and from the Command Uplink process. Messages received from RMS, TLM, and CMD:Uplink; ground telemetry queries from FUI, and time-outs (rel. B) are processed and then the controller returns to SuperWait, returning to its previous state. Commands, operator prompt responses, and load requests (rel. B) from FUI cause a change of state within or upon return to SuperWait.

In the diagram, processing which does not affect the SuperWait substate is shown at the bottom of the diagram. Processing of messages which do affect the substate are shown within, above, and to the right of the SuperWait box. Processing of commands begins with "receive command" and continues toward the right. As can be seen, during command processing while possible prompt responses are being waited upon, the controller returns to a wait state and other messages can be processed.

Some things were omitted from the diagram for the sake of simplicity. If an invalid directive is received from FUI (e.g., if a command is received while the controller is in "wait for critical prompt response" state), an error is logged, the directive is ignored, and the state does not change. If prerequisite checking is turned off, the "validate command" state will either transition to "wait for critical prompt" response or to "build and send command", depending on whether the command is critical. Activities within states are generally not listed; state names should be self-explanatory within the context of the rest of the subsystem documentation.

The exit state is reached when a Shutdown directive is received from the RMS.

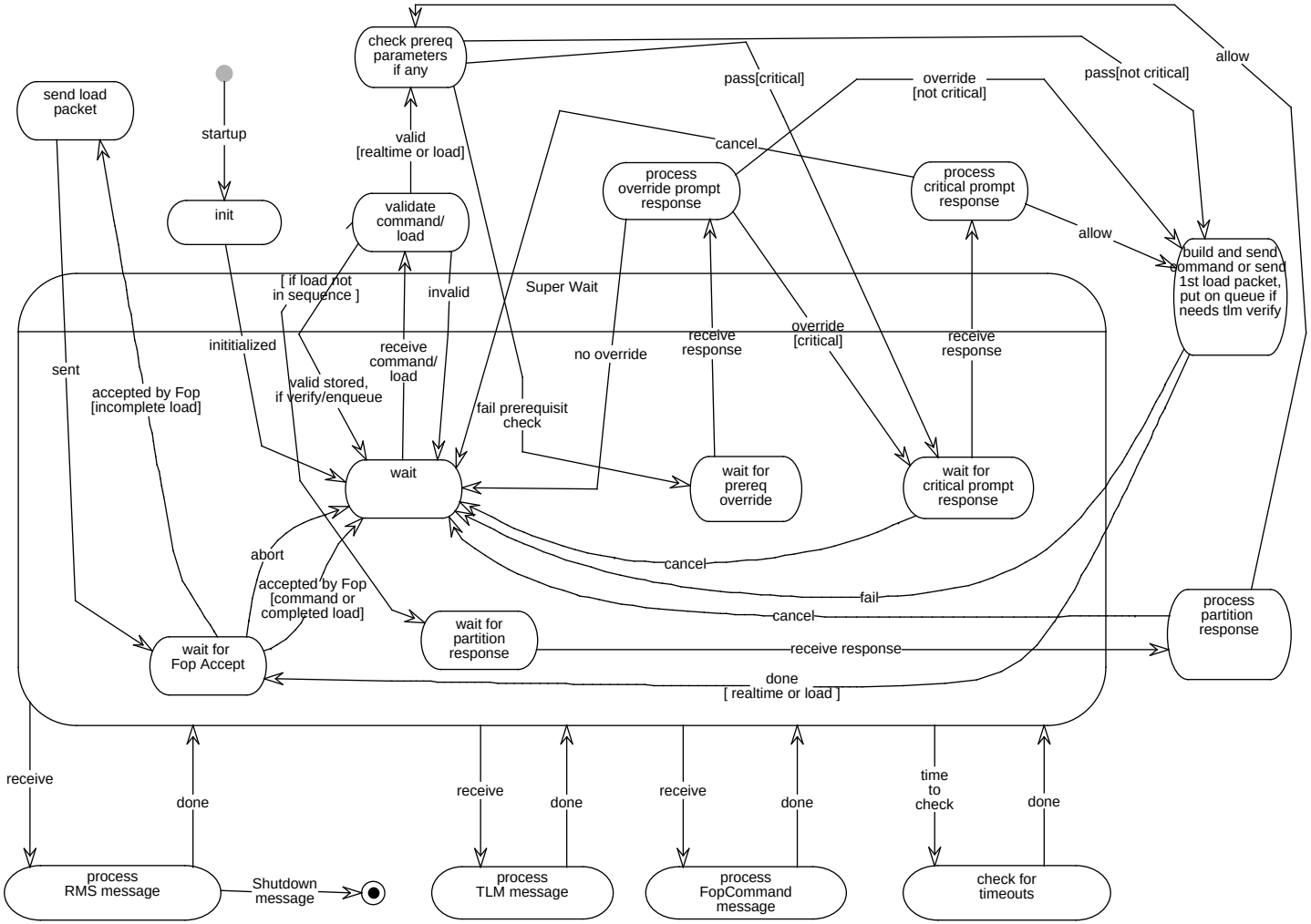

Figure 3.2.4.27-1. FcCdCmdController state diagram

3.2.4.28 FcCdRtCmd State Diagram Description

During its creation by the FcCdCmdController, the FcCdRtCmd attributes are initialized. The FcCdRtCmd object then enters the waiting mode. Upon receiving the validate command from the FcCdCmdController, it will perform submnemonics validation. If any submnemonic is not validated, it will return a Fail status to the FcCdCmdController. If all submnemonics are validated, it then enters the CheckPrerequisite state. There are two scenarios in this state. First, if there is no prerequisite check required or the check is successful, it then moves on to the CheckCritical state. In the second scenario, the prerequisite check is unsuccessful; the FcCdRtCmd then prompts for override and enters the Wait-for-Override-Response state. Upon receiving the response (asynchronously), it enters the CheckCritical state if the response is to ALLOW; otherwise, it returns fail status and exits.

Inside the CheckCritical state, if the command is not critical, the FcCdRtCmd object returns validation success status and then enters the Wait-for-Build-Command state. If the command is critical, it prompts for critical permission and enters the Wait-for-Permission state. Upon receiving the response (asynchronously), if the response is to ALLOW, it enters the Wait-for-Build-Command state; otherwise, it returns fail status and exits.

In the Wait-for-Build-Command state, upon receiving the Build command from the FcCdCmdController, the FcCdRtCmd object constructs the binary command in the 1553-B format and then sends this binary command to the uplink process. It returns the status and exits.

3.2.4.29 FcCdLoadCmd State Diagram Description

After being instantiated by the Command Controller, the FcCdLoadCmd object enters the wait for validation state. Upon receiving validation request from the controller, the FcCdLoadCmd object enters the Validation state, where it performs the following:

It instantiates an FcCdLoadData object;

It invokes the Init() function of the FcCdLoadData object to read in the header file and the load data;

It invokes the ValidateLoadParameters() of the FcCdLoadData to validate the spacecraftId, the destination and time window for the current load;

If the validation is unsuccessful, the FcCdLoadCmd returns Unsuccessful status then exits. Otherwise, it checks for load criticality. If the load is critical, it prompts for critical response, returns control to the command controller and enters the Wait For Critical Response state. If the load is not critical, it returns successful validation status and enters the Wait to be Sent state.

Inside the Wait for Critical Response state, upon receiving the ALLOW response, it returns Success status to the controller and enters the Wait to be Sent state. If the response is CANCEL, it returns Unsucessful status and exits.

In the Wait to be Sent state, upon receiving the send command from the controller, it sends out one packet from the load, returns status to the controller. If the packet is the last one in the load, it exits; otherwise, it goes back to the Wait to be Sent state.

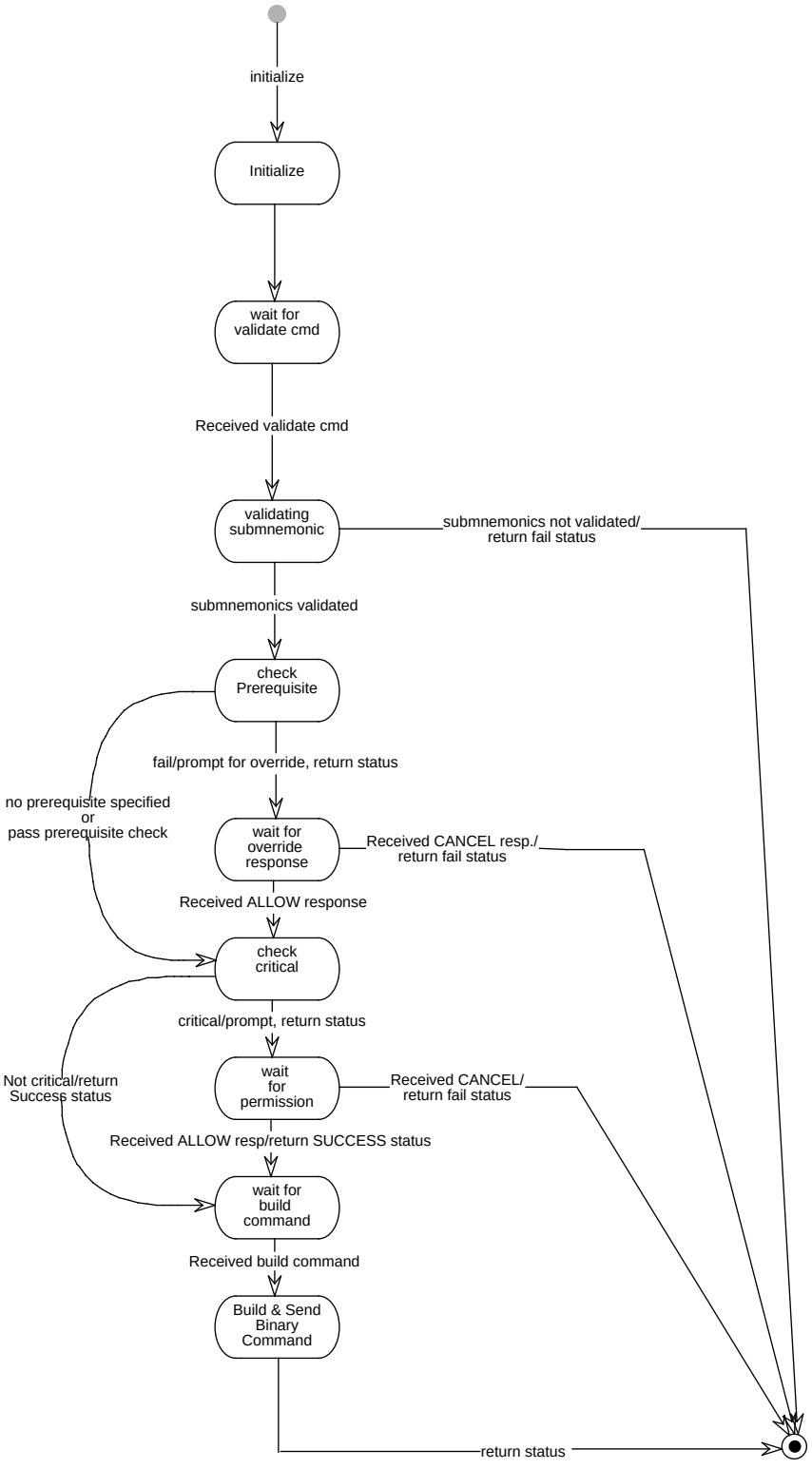

Figure 3.2.4.28-1. FcCdRtCmd state diagram

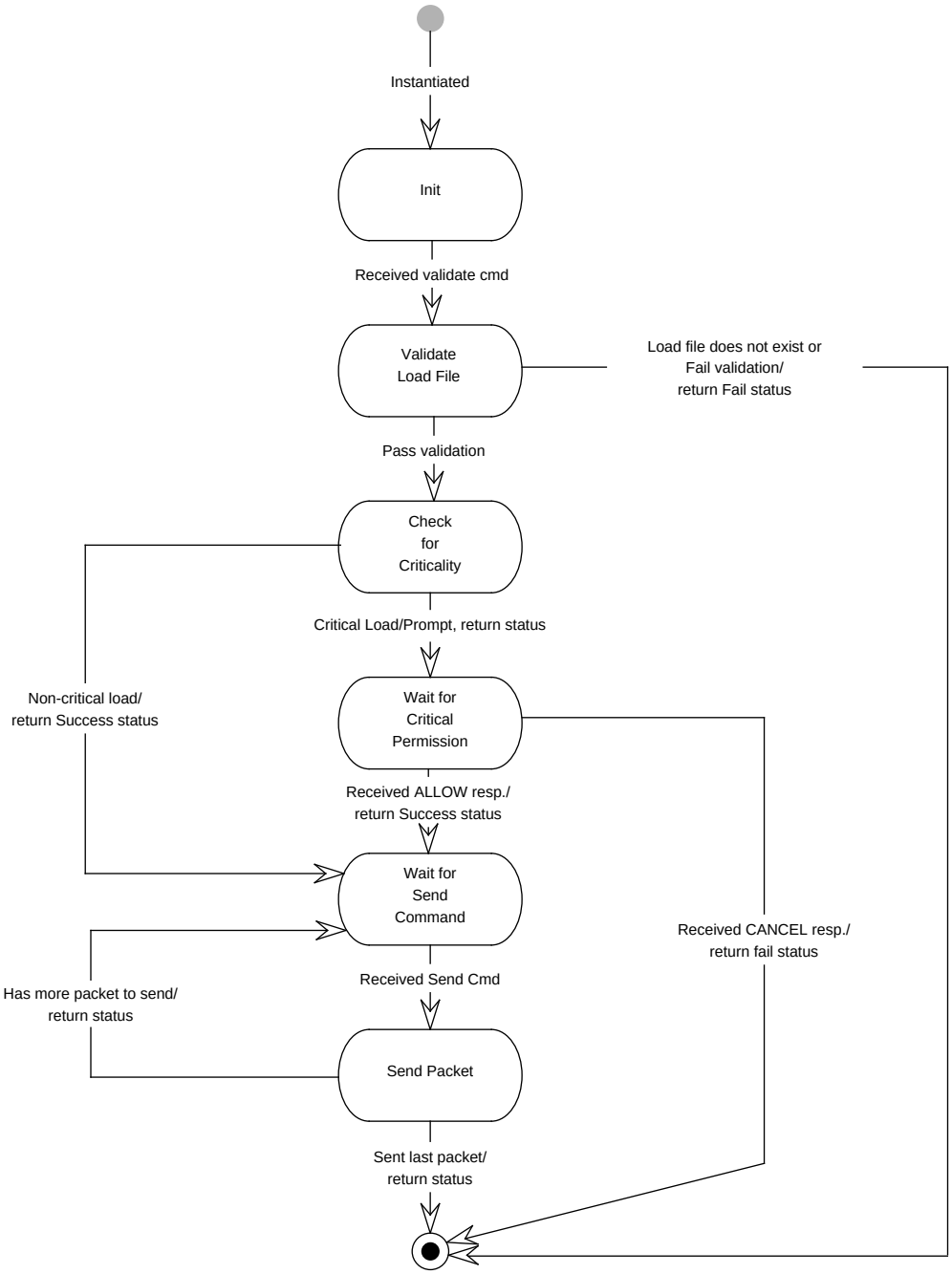

Figure 3.2.4.29-1. FcCdLoadCmd state diagram

3.2.5 FormatCommand Data Dictionary

FcCdCmdController

class **FcCdCmdController**

This class contains all the attributes that characterize the state of the Format task. It also contains the functions which perform the basic housekeeping of the process and coordinate the activities of the other objects.

Public Functions

EcTBoolean **Init**()

Run

FcCdFopFormatProxy

class **FcCdFopFormatProxy**

Private Data

FoGnFopAcceptMsg **myAcceptMsg**

sent to FormatCommand each time a command is accepted

FoGnFopReceiptMsg **myReceiptMsg**

sent to FormatCommand each time a command is receipt confirmed by s/c or times out

FcCdFormatRcvIf

class **FcCdFormatRcvIf**

receives all asynchronous messages that come in across the ipc. Because of continuing developments regarding the ipc, this interface is still somewhat undetermined.

FcTCdStatus

enum **FcTCdStatus**

Enumerators

FcECdPromptingForCritPermit
FcECdPromptingForPrereqOverride
FcECdSuccessfulValidation
FcECdWrongOrder

FcGnFopAcceptMsg

class **FcGnFopAcceptMsg**

This message class carries a message which indicates if FopCommand process is ready for the next command.

Base Classes

public **FcGnFopMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual function FoGnGenericMsg::Execute(). It calls FcCdCmdController::ProcessFopAcceptMsg member function.

FcGnFopCmdMsg

class **FcGnFopCmdMsg**

This message class is sent to FopCommand process. It contains a real time command in 1553B format.

Base Classes

public **FcGnFopDataMsg**

Public Functions

EcTVoid **Execute**()

This operation knows how to invoke ProcessNewCmd function of FopCommand process.

Private Data

EcTBoolean **myBcFlag**

This attribute identifies if the current command is a CCSDS control command.

FcGnFopDataMsg

class **FcGnFopDataMsg**

This message class is the base class for FcGnFopCmdMsg and FcGnFopPacketMsg.

Base Classes

public **FcGnFopMsg**

Private Data

EcTUChar* **myData**

This attribute identifies the binary command data.

EcTInt **myDataLength**

This attribute identifies the length of the command data.

FcGnFopMsg

class **FcGnFopMsg**

abstract class that represents all messages passed from FormatCommand to FopCommand and vice-versa

Base Classes

public **FoGnGenericMsg**

Private Data

EcTInt **mySeqNum**

used as method of making each command unique for reference

FcGnFopPacketMsg

class **FcGnFopPacketMsg**

This message class is sent to FopCommand process. It carries a CCSDS packet.

Base Classes

`public` **`FcGnFopDataMsg`**

Public Functions

`EcTVoid` **`Execute`**`()`

This operation knows how to invoke ProcessNewPacket function of FopCommand process.

Private Data

`FcTCdLoadStage` **`myLoadStage`**

This attribute identifies the memory load stage. i.e. first or last partitin etc.

FcGnFopReceiptMsg

`class` **`FcGnFopReceiptMsg`**

This message class carries a message which indicates if a command is CLCW verified.

Base Classes

`public` **`FcGnFopMsg`**

Public Functions

`EcTVoid` **`Execute`**`()`

This operation overrides the virtual function FoGnGenericMsg::Execute(). It calls FcCdCmdController::ProcessFopReceiptMsg member function.

Private Data

`EcTBoolean` **`mySuccess`**

This attribute identifies if the command is cuccessfully CLCW verified.

FoGnCmdFuiIf

`class` **`FoGnCmdFuiIf`**

Public Functions

`EcTVoid` **`Error`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that an error occured concerning this command

`EcTVoid` **`GoAhead`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that it can send another command

`EcTVoid` **`Init`**`()`

Critical This operation sends FUI a FoUiStatus message that this command needs a critical prompt

`EcTVoid` **`PartitionOrder`**`(EcTInt seqNum)`

ReceiptVerifyPass This operation sends FUI a FoUiStatus message that this command was received by the S/C

`EcTVoid` **`PrereqFail`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that this command failed prereq check and FormatCommand is awaiting an override or cancel

`EcTVoid` **`PrereqPass`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that this command passed prereq check

`EcTVoid` **`PutResponse`**`(RWCString string)`

records a message to be displayed by Fui accompanying the next status sent

`EcTVoid` **`ReceiptVerifyFail`**`(EcTInt)`

TlmVerifyPass This operation sends FUI a FoUiStatus message that this command has been executed on board the S/C

`EcTVoid` **`TlmVerifyFail`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that this command failed to execute

`EcTVoid` **`TlmVerifyNone`**`(EcTInt)`

This operation sends FUI a FoUiStatus message that this command has no tlm verify parameters

FoGnFormatRmsIf

`class` **`FoGnFormatRmsIf`**

Public Functions

`EcTBoolean` **`SendReceipt`**`()`

sends a FoGnRmsReceiptMsg to RMS:String Manager task

Private Data

`FoGnRmsReceiptMsg` **`myReceiptMsg`**

FoGnFormatTlmIf

`class` **`FoGnFormatTlmIf`**

Public Functions

`EcTVoid` **`Init`**`(ipc info)`

PrepareForDump sends TLM subsystem a message notifying it of an imminent dump

FoGnFuiAbortLoadMsg

`class` **`FoGnFuiAbortLoadMsg`**

This message class carries a abort load message to the FormatCommand process.

Base Classes

`public` **`FoGnFuiMsg`**

Public Functions

`EcTVoid` **`Execute`**`()`

This operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::ProcessLoadAbort function.

FoGnFuiCmdProxy

`class` **`FoGnFuiCmdProxy`**

FoGnFuiCriticalRspMsg

`class` **`FoGnFuiCriticalRspMsg`**

this message class carries a response to a critical prompt to the FormatCommand process

Base Classes

public **FoGnFuiMsg**

Public Functions

EcTVoid **Execute**()

this operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::ProcessCriticalRsp().

Private Data

FcTCdCriticalRsp **myResponse**

This attribute identifies the response.

FoGnFuiLoadMsg

class **FoGnFuiLoadMsg**

This message class carries a process Load message to FormatCommand process.

Base Classes

public **FoGnFuiMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::ProcessLoadRequest function.

Private Data

RWCString **myName**

This attribute identifies the load name.

FcTCdLoadType **myType**

This attribute identifies the load type.

FoGnFuiMsg

class **FoGnFuiMsg**

this abstract class represents all messages passed from FoGnFuiCmdProxy to FcCdRcvIf

Base Classes

public **FoGnGenericMsg**

Private Data

EcTInt **mySeqNum**

used to give each issued command a unique number

EcTInt **myUserId**

gives the used id of the user sending the message

EcTInt **myWksId**

gives the workstation id of the console the command is sent from

FoGnFuiPartRspMsg

class **FoGnFuiPartRspMsg**

This message class carries a response to a prompt for a out of sequence load.

Base Classes

public **FoGnFuiMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual function FoGnGenericMsg::Execute(). It calls FcCdCmdController::ProcessPartitionRsp member function.

Private Data

FcTCdRsp **myResponse**

This attribute identifies the response.

FoGnFuiPrereqRspMsg

class **FoGnFuiPrereqRspMsg**

This message class carries a response to a prerequisite check prompt to the FormatCommand process.

Base Classes

public **FoGnFuiMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::ProcessPrereqRsp().

Private Data

FcTCdRsp **myResponse**

This attribute identifies the response.

FoGnFuiStoredCmdMsg

class **FoGnFuiStoredCmdMsg**

This message class carries a stored command to FormatCommand process.

Base Classes

public **FoGnFuiMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual function FoGnGeneric::Execute(). It calls FcCdCmdController::ProcessStoredCommand member function.

Private Data

FcTCdSource **mySource**

This attribute identifies the stored command source.

RWCString **myString**

This attribute identifies the stored command string.

FoGnGenericMsg

class **FoGnGenericMsg**

abstract class represents all messages sent to FormatCommand or sent by FormatCommand to FopCommand. These messages know how to execute themselves.

Base Classes

public **RWCollectable**

Public Functions

virtual EcTVoid **Execute**(void)

is overridden in each concrete class so that each message knows what operation in FcCdCmdController to call in order to process itself

FoGnRmsFormatInitMsg

class **FoGnRmsFormatInitMsg**

message is sent from RMS to FormatCommand during initialization to set configuration attributes.

Public Functions

EcTVoid **Execute**(void)

Private Data

EcTInt **myDbId**

initializes value

FoTGnAddress **myFopAddr**

initializes value

FoTGnAddress **myFuiAddr**

initializes value

FcTCdOperationMode **myOperationMode**

initializes value

FoTGnAddress **myParamServerAddr**

initializes value

FcTCdPrimaryMode **myPrimaryMode**

initializes value

EcTInt **myScId**

initializes value

FoTGnAddress **myTlmAddr**

initializes value

FoGnRmsFormatPrimaryModeMsg

class **FoGnRmsFormatPrimaryModeMsg**

This message class carries a "set mode" request to FormatCommand process.

Base Classes

public **FoGnRmsMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual function FoGnGenericMsg::Execute(). It calls FcCdCmdController::SetMode member function.

FcTCdPrimaryMode **GetPrimaryMode**()

This operation returns the mode of the current string to the caller.

EcTVoid **SetPrimaryMode**(FcTCdPrimaryMode)

This operation sets the mode of the current string.

Private Data

FcTCdPrimaryMode **myPrimaryMode**

This attribute identifies the mode of current string.

FoGnRmsFormatProxy

class **FoGnRmsFormatProxy**

FoGnRmsFormatShutdownMsg

class **FoGnRmsFormatShutdownMsg**

This message class carries a shutdown request to FormatCommand process.

Base Classes

public **FoGnRmsMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::Shutdown function.

FoGnRmsMsg

class **FoGnRmsMsg**

abstract class represents all messages sent from FoGnRmsFormatProxy to FcCdRcvIf

Base Classes

public **FoGnGenericMsg**

FoGnRmsSaveFormatSnapshotMsg

class **FoGnRmsSaveFormatSnapshotMsg**

This message class carries RMS request "Save snap shot" to FormatCommand process.

Base Classes

public **FoGnRmsMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual FoGnGenericMsg::Execute() function. It calls FcCdCmdController::SaveSnapshot function.

EcTVoid **FoGnRmsSaveSnapshotMsg**()

This is the default constructor.

Private Data

RWCString **myFileName**

This attribute identifies my snap shot file name.

FoGnRmsSetCmdAuthUserMsg

class **FoGnRmsSetCmdAuthUserMsg**

This message class carries a "Set command authorized user" request to FormatCommand process.

Base Classes

public **FoGnRmsMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual function FoGnGenericMsg::Execute(). It calls FcCdCmdController::UpdateCmdAuthUser function.

Private Data

EcTInt **myUserId**

This attribute identifies the user id.

EcTInt **myWksId**

This attributes identifies the work station id.

FoGnRmsSetPrereqCheckMsg

class **FoGnRmsSetPrereqCheckMsg**

This message class carries RMS request "set prerequisite check state" to FormatCommand process.

Base Classes

public **FoGnRmsMsg**

Public Functions

EcTVoid **Execute**()

This operation overrides the virtual FoGnGenericMsg::Execute(), calling FcCdCmdController::SetPrereqCheckState function.

Private Data

FcTCdPrereqCheckState **myPrereqCheckState**

This attribute identifies the new prerequisite check state.

FoGnTlmDumpMsg

class **FoGnTlmDumpMsg**

Base Classes

public **FoGnTlmMsg**

Private Data

EcTBoolean **myAbsoluteFlag**

indicates if the dump is an absolute dump

EcTInt **myAddress**

indicated the address of the beginning of the dump

EcTInt **mySegOffset**

indicates the segment offset of the dump

EcTInt **myTableId**

indicates the table ID of the table being dumped

EcTInt **myWordLength**

indicates the length of the dump in words

FoGnTlmMsg

class **FoGnTlmMsg**

abstract class represents all the messages sent from FoGnFormatTlmIf to FoGnTlmDumpProxy

Base Classes

public **RWCollectable**

FoUiStatus

class **FoUiStatus**

message passed from FormatCommand to FUI giving status of a directive

Private Data

EcTInt **mySeqNum**

provides a unique identifier for each directive

FoTUiStatus **myStatus**

provides the status of the directive

RWCString **myText**

provides text which may be displayed to the user to accompany the status.

FcCdBaseCmd

class **FcCdBaseCmd**

This is the base class for all command type: real-time, hex/binary, stored or load.

Public Construction

FcCdBaseCmd(void)

This member function is the default constructor.

~FcCdBaseCmd(void)

This member function is the destructor.

Public Functions

virtual EcTBoolean **CheckCritical**(void)

This member function is a virtual function.

virtual EcTInt **ProcessCriticalRsp**(FoGnFuiCriticalRspMsg* msg)

This member function is a virtual function.

virtual EcTBoolean **Validate**(void)

This member function is a virtual function.

virtual EcTBoolean **VerifyTlm**(Struct_Pid* TlmMsg)

This member function is a virtual function.

Private Data

static FoGnCmdCmsProxy* **myCmsProxy**

This member variable points to Cms proxy.

static FoDsFile* **myDsFile**

This member variable points to the FoDsFile proxy

static FdEvEventLogger* **myEventLogger**

This member variable points to FdEvEventLogger proxy.

static FoGnCmdFopProxy* **myFopProxy**

This member variable points to the FopCommand proxy.

EcTInt **myFuiCId**

myFuiCid

This member variable contains the Fui Cmd Id.

enum **myFuiCmdType**

static FoGnCmdFuiProxy* **myFuiProxy**

This member variable points to Fui Proxy.

static FoGnCmdTlmProxy* **myTlmProxy**

This member variable points to Tlm proxy.

Private Types

enum

This member variable contains the command type.

Enumerators

CAC
script
stored

FcCdCmd

class **FcCdCmd**

A base class representing either a real-time command or a stored command.

Public Construction

FcCdCmd(command_struct* cmd_str)

object constructor.

~FcCdCmd(void)

object destructor.

Public Functions

EcTVoid **SetTime**()

RecordTime

This member function set myEnqueueTime to current time.

virtual EcTBoolean **TimeOut**()

This member function is a virtual function

EcTBoolean **VerifyTlm**(Struct_Pid* TlmMsg)

This member function is used for telemetry verification

Private Data

EcTInt **myCmdLen**

the length of the command, in words.

enum **myCmdType**

EcTBoolean **myCritical**

This is a Boolean that indicates whether or not the operator is to be prompted prior to processing the command. If prompting is required, the operator must respond positively before the command is processed; a negative response will prevent uplink of the command.

structure **myDescriptor**

This attribute is information about the command, such as command type and data length.

structure **myDestination**

This attribute is information pertaining to the routing of a command once on board the spacecraft, i.e., the instrument of the subsystem to which the command is directed.

set **myExpectedTlmValue**

This attribute is a set of expected values for the telemetry parameters in myTlmVerifyPid set to determine command execution verification. There is one pair of high/low values for each (1 to 4) of myTlmVerifyPid.

RWCString **myMnemonic**

This is the command id corresponding to the mnemonic command, as referenced from within the FUI Subsystem.

EcTInt **myNumFixedDataRec**

This attribute identifies number of fixed data records in the database file.

EcTInt **myNumPrereq**

This attributes identifies number of prerequisite records in the database file.

EcTInt **myNumVarDataRec**

This attribute identifies the number of variable data records in the database file.

enumerated **mySubSystem**

This attribute idnetifies the name of subsystem to which the command is directed.

EcTInt **myTlmVerifyPid**

This is telemetry parameter used to verify the command using telemetry.

`EcTReal` **`myVerifyWaitInterval`**

This is the amount of time, in miliseconds, to wait before declaring a command as having failed verification.

`array` **`of`**

This is a record containing information for prerequisite checking: telemetry PID, prerequisite type (raw or converted), and the upper / lower limit values to define the range of accepted values for the prerequisite check.

`array` **`of`**

This attribute identifies a set of command fixed data record (if existed).

Private Types

`enum`

command type (e.g. BDU Relay Drive/Logic/Serial/No-op)

Enumerators

`Bdu`
`CCSDSBc`
`CtiuLoad`
`Dump`
`Logic`
`NOOP`
`Relay`
`SCCLoad`
`Serial`

FcCdCmdDef

`class` **`FcCdCmdDef`**

This class contains all information related to a command.

Public Construction

`FcCdCmdDef``(command_struct* cmd_str)`

This member function is the default constructor.

`~FcCdCmdDef``()`

This member function is the destructor.

Public Functions

`EcTVoid` **`GetCmRecord`**`(command_struct* cmd_str)`

GetCmdRecord

This member function returns all attributes of this object.

`EcTInt` **`hash`**`()`

This member function returns hash number for this object. The hash number is used to insert the object into RWSet.

`RWBoolean` **`isEqual`**`(const RWCollectable* cmd)`

This member function defines the meaning of equivalence for two objects of this class.

`EcTVoid` **`restoreGuts`**`(RWvistream& strm)`

This member function restores an object from a "flat" definition.

`EcTVoid` **`saveGuts`**`(RWvistream& strm)`

This member function saves an object to a stream so that it can be restored late by restoreGuts function.

Private Data

`EcTInt` **`myCmdLen`**

This member variable contains the length of the command in words.

`enumerated` **`myCmdType`**

This member variable contains the command type (Relay,Logic,Serial,NOOP, CtiuLoad, SCCLoad, CCSDSBc, Dump)

`EcTBoolean` **`myCritical`**

This member variable is a Boolean flag that indicates whether or not the operator is to be prompted prior to processing the command. If prompting is required, the operator must respond positively before the command is processed; a negative response will prevent uplink of the command.

`set` **`myExpectedTlmValue`**

This attribute identifies a set of expected values for the telemetry parameters in myTlmVerifyPid set to determine command execution verification. There is one pair of high/low values for each (1 to 4) of myTlmVerifyPid.

`RWCString` **`myMnemonic`**

This attribute identifies the number of fixed data records in the database file.

`EcTInt` **`myNumPrereq`**

This attribute identifies the number of prerequisite records in the database file.

`EcTInt` **`myNumVarDataRec`**

This attribute identifies the number of variable data records in the database file.

`enumerated` **`mySubSystem`**

This attribute identifies the name of subsystem to which the command is directed.

`set` **`myTlmVerifyPid`**

This attribute is a set (1 to 4) of telemetry parameters used to verify the command using telemetry.

`EcTReal` **`myVerifyWaitInterval`**

This attribute identifies the amount of time, in miliseconds, to wait before declaring a command as having failed verification.

`array` **`of`**

`array` **`of`**

This attribute a record containing information for prerequisite checking: telemetry PID, prerequisite type (raw or converted), and the upper / lower limit values to define the range of accepted values for the prerequisite check.

`array` **`of`**

This attribute identifies a set of command fixed data record (if existed).

FcCdCommandDatabase

`class` **`FcCdCommandDatabase`**

This class provides database information needed for all aspects of command processing, such as validation, building and verifcation. This class is derived from Rogue Wave RWSet class.

Public Construction

`FcCdCommandDatabase``()`

This member function is the default constructor.

`~FcCdCommandDatabase``()`

This member function is the destructor.

Public Functions

`EcTInt` **`GetCmdDefinition`**`(command_struct* cmd_str)`

This member function gets the information necessary to build a command, returns all information in the database related to the given command mnemonic.

`EcTBoolean` **`Init`**`(FoDsFile, DatabaseId)`

This member function loads information from the database into the hash table.

`EcTVoid` **`restoreGuts`**`(RWvistream& strm)`

saveGuts

This member function is a Rogue Wave function; it stores an object into a file so that the object can be later restored using restoreGuts.

`EcTVoid` **`restoreGuts`**`(RWvistream& strm)`

This member function is a Rogue Wave function; it restores an object from a "flat" definition.

FcCdHexCmd

class **FcCdHexCmd**

This class is used for hex/binary command in 1553-b format.

Public Construction

`FcCdHexCmd``(void)`

This member function is the default constructor.

`~FcCdHexCmd``(void)`

This member function is the destructor.

Public Functions

`EcTBoolean` **`ProcessCriticalRsp`**`(FoGnFuiCriticalRspMsg* FuiRspMsg)`

This member function handles the response to critical prompt

`EcTBoolean` **`SendCmd`**`(void)`

This member function sends the hex cmd to FopCommand

Private Data

`RWCString*` **`myBinaryCmd`**

This member variable contains the hex command.

FcCdLoadCmd

class **FcCdLoadCmd**

This class is used for all load command.

Base Classes

`public` **`FcCdRtCmd`**

Public Construction

`FcCdLoadCmd``(RWCString* spacecraftId, RWCString* LoadId)`

This member function is the constructor.

`~FcCdLoadCmd``(void)`

This member function is the destructor.

Public Functions

`EcTBoolean` **`ProcessCriticalRsp`**`(FoGnFuiCriticalRspMsg* FuiMsg)`

This member function is used to process critical response.

`EcTBoolean` **`ProcessPartitionRsp`**`(FoGnFuiPartRspMsg* FuiMsg)`

This member function is used to process partition response.

`EcTInt` **`SendLoad`**`(void)`

This member function is used to send a load to FopCommand process, one packet at a time.

`FcTCdStatus` **`Validate`**`(FoUiInstruction* FuiMsg)`

This member function is used to validate a load; such item such as spacecraftId and time window will be validated.

`EcTBoolean` **`VerifyTlm`**`(Struct_Pid* TlmMsg)`

This member function is used for telemetry verification of a load.

Private Data

`RWCString*` **`myCRC`**

This member variable contains the CRC of the load.

`FcCdLoadData*` **`myLoadData`**

This member variable points to the load data (e.g. packets).

`RWCString*` **`myLoadId`**

This member variable contains the Load Id.

`RWCString*` **`mySpacecraftId`**

This member variable contains the spacecraft ID.

FcCdLoadData

class **FcCdLoadData**

class FcCdLoadData; This class contains the file header and data for a load.

Public Construction

`FcCdLoadData``(myLoadId, myDsFile, myEventLogger, myFopProxy, myFuiProxy)`

This member function is the constructor.

`~FcCdLoadData``(void)`

This member function is the destructor.

Public Functions

`EcTVoid` **`GetLoadParameters`**`(RWCString* CRC, EcTBoolean* Critical, EcTInt* TlmPid, EcTInt* WaitInterval)`

This member function returns important attributes concerning the load: CRC, critical flag, TlmPid and VerifyWaitInterval.

`EcTBoolean` **`Init`**`(void)`

This member function is used to read load data and header.

`EcTInt` **`SendPacket`**`(void)`

This member function sends one packet out to FopCommand process.

`EcTBoolean` **`ValidateLoadParameters`**`(RWCString* SpacecraftId)`

This member function validates parameters of a load (spacecraftId and time window)

Private Data

`RWCString*` **`myCRC`**

This member variable contains the CRC of a load.

`EcTBoolean` **`myCritical`**

This member variable indicates the criticality of the load.

`EcTInt` **`myCurrentPacket`**

This member variable contains the number of the packets that have been sent out.

`RWCString*` **`myData`**

This member variable contains the load packets.

`RWCString*` **`myDestination`**

This member variable contains the destination for the load.

`FoDsFile*` **`myDsFile`**

This member variable points to the FoDsFile proxy.

`FdEvEventLogger*` **`myEventLogger`**

This member variable points to FdEvEventLogger proxy.

`FoGnCmdFopProxy*` **`myFopProxy`**

This member variable points to the FopCommand proxy.

`EcTInt` **`myFuiCid`**

This member variable contains the Fui command Id.

`enum` **`myFuiCmdType`**

`FoGnCmdFuiProxy*` **`myFuiProxy`**

This member variable points to the Fui Proxy.

`EcTInt` **`myLoadDataIndex`**

This member variable contains the "mark" for the start of the next packet in the load.

`RWCString*` **`myLoadId`**

This member variable contains the Load Id.

`RWCString*` **`mySpacecraftId`**

This member variable contains the spacecraft id.

`EcTInt` **`myTlmPid`**

This member variable contains the load Tlm Pid.

`EcTInt` **`myTotalPacket`**

This member variable contains the total number of packets in the partition.

`EcTInt` **`myWaitInterval`**

This member varable contains the relative time window for load tlm verified.

`time*` **`myWindow`**

This member variable contains the time window for the load to be uplinked

Private Types

`enum`

This member variable contains the Fui Command Type.

Enumerators

CAC
script
stored

FcCdRtCmd

class **FcCdRtCmd**

(type of FcCdCmd) A derived class representing a real-time command that has been issued.

Base Classes

public **FcCdCmd**

Public Construction

FcCdRtCmd(command_struct* cmd_str)

This is the object constructor.

~FcCdRtCmd()

This is the object destructor.

Public Functions

EcTBoolean **Build**(FopProxy, FdEvEventLogger, FuiProxy)

This function formats the command into spacecraft command format using the BinaryCmd.

EcTBooleam **CheckCritical**(FuiProxy)

This member function checks myCritical value and, if true, prompts the user for an allow/cancel response.

EcTBoolean **CheckPrereq**(TlmProxy, FdEvEventLogger)

This member function checks the Prerequisite telemetry point(s), if any, unless myPrerequisiteStatus is disabled. CheckTlmValue is used to compare the actual telemetry values against their prerequisite values. If the return status indicates failure, the user is prompted for an override/ cancel response.

EcTInt **ProcessCriticalRsp**(FdEvEventLogger, FuiProxy, FuiMsg)

This member function processes the response from user to earlier prompt for critical command permission.

EcTInt **ProcessPrereqRsp**(FdEvEventLogger, FuiProxy, FuiMsg)

This member function processes the response from user to earlier prompt for prerequisite override.

EcTBoolean **SendCmd**(FopProxy)

This member function forwards the commands and directives to FcCmCCSDSFop. The command is in 1553-B (AM-1 real-time) format.

EcTInt **Validate**(PrereqFlag, FuiProxy, TlmProxy, FdEvEventLogger, FuiCmdStatus)

performs the following functions: checks submnemonics for validity checks prerequisite state(s) by invoking CheckPrereq and prompts for Prerequisite override if fail prerequisite check invokes critical prompt for critical commands by invoking CheckCritical.

Private Data

structure **myBinaryCmd**

This attribute is the digital representation of the command.

structure **myCmdDirective**

This is a structure contains the parsed command directive.

`EcTBoolean` **`myPrerequisiteStatus`**

This attribute reflects the status of prerequisite checking.

FcCdStoredCmd

class **FcCdStoredCmd**

This class is used for stored commands.

Base Classes

`public` **`FcCdCmd`**

Public Construction

`FcCdStoredCmd``(void)`

This member function is the default constructor.

`~FcCdStoredCmd``(void)`

This member function is the destructor.

Public Functions

`EcTBoolean` **`TimeOut`**`(void)`

This member function returns the status of whether or not the current time is outside or inside the waiting window.

Private Data

`static time` **`myDownLinkDelay`**

This member variable contains the time delay for downlink.

3.3 FopCommand Description

The Command Fop process is responsible for taking commands built by the FormatCommand process, wrapping them in the appropriate header and footer information, and sending them to the TransmitCommand process to be sent to the satellite via EDOS. The Fop process must also accept a small number of directives from RMS, as well as process Command Link Control Words or CLCWs received from the satellite through EDOS which keeps it up to date on which commands have been received onboard.

3.3.1 FopCommand Context Description

The FopCommand process receives formatted commands and memory loads from the FormatCommand process, puts the appropriate wrappers on them according to CCSDS standard and builds CLTUs. FopCommand process then sends the formatted data on to the TransmitCommand process as CLTUs. The receipt status of a uplinked command will be reviewed by a down linked CLCW forwarded to FopCommand by EDOS. When the receipt of one of these is confirmed by the spacecraft, FopCommand sends a command receipt to the FormatCommand process. The Resource Management Subsystem (RMS) sends config info, reconfig requests and snapshot request to FopCommand, and receives status messages from it. The config info includes the spacecraft ID, the database ID, the state of the current string (i.e. backup or primary), the mode of current string (i.e. real-time or simulation), the address of parameter server, the address of FormatCommand process and the address of TransmitCommand process. FopCommand process also sends ground telemetry updates to Parameter Server when ever a ground telemetry is changed either by a RMS directive or a CLCW. The ground telemetry parameter set may include the following parameters: the current state of the Fop protocol, the wait flag of the current CLCW, the Lockout flag of the current CLCW, the retransmit flag of the current CLCW, the current frame sequence number (V(S)), the next expected acknowledgment frame sequence number (NN(R)), the current timer initial value (T1_Initial), the transmission limit, the current transmission count and the Fop Sliding Window width etc. The FOS Data Management Subsystem (DMS) is used to save and retrieve a file containing config info, and to log events.

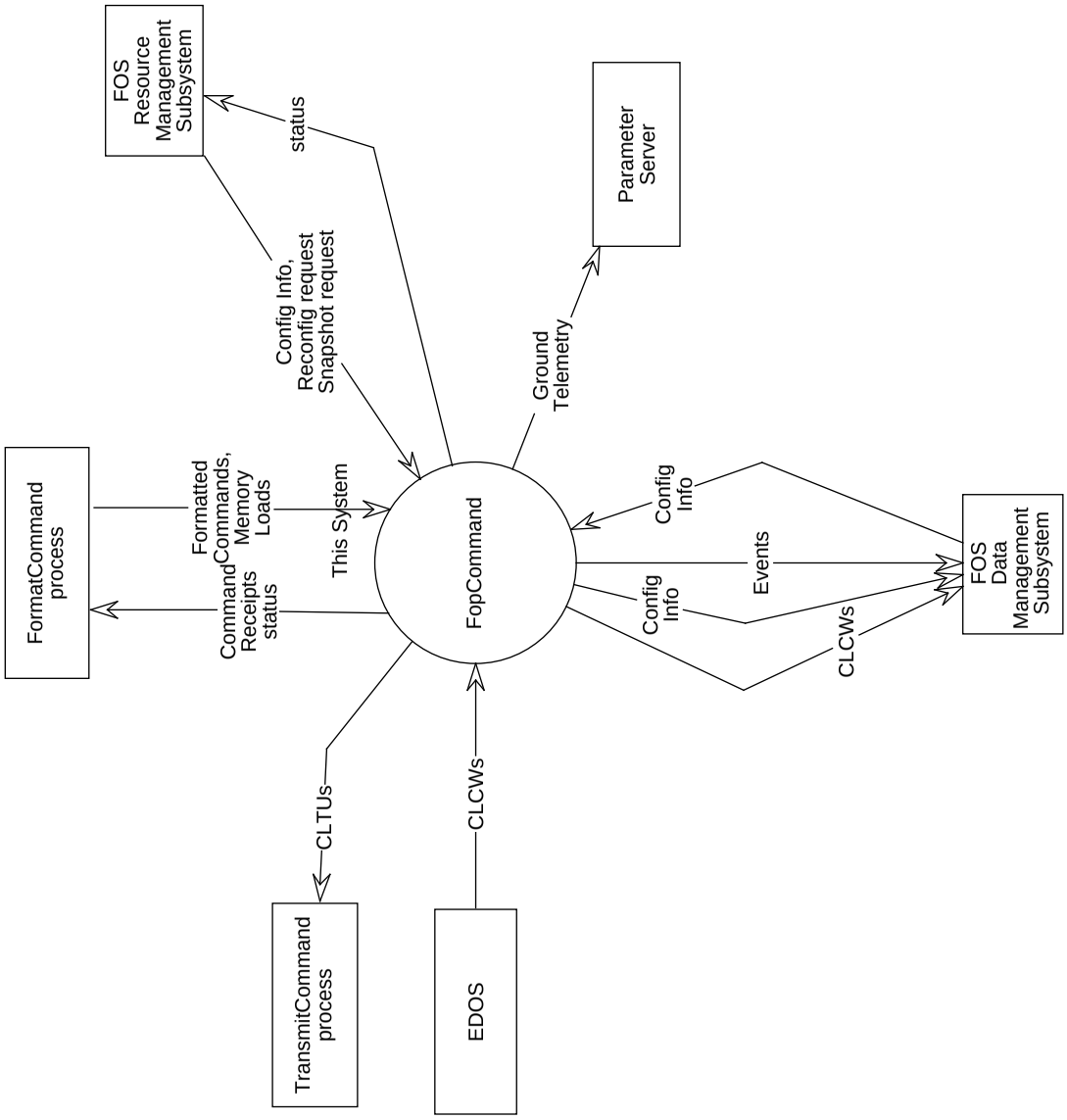

Figure 3.3.1-1. FopCommand Context Diagram

3.3.2 FopCommand Interfaces

***Table* 3.3.2. FopCommand Interfaces**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Recieve CLCWs from EDOS	FoGnCmd Ground StationIF	Recieve binary telemetry from EDOS	CMD: Fop	CMD: Fop	once per second
Send CLTUs	FoGn CmdFop Transmit Proxy	Sends CLTUs to the Transmit Command process	CMD: Transmit	CMD: Fop	once per command
Send ack-nowledge-ments	FcGnTcCltu FoGnFop Format Proxy	Message class for a CLTU Sends acknow-ledgments to the Format Command process	CMD: Format	CMD: Fop	twice per command
Receive commands and directives	FoGnCmd FopFormat IF	Receives Cmds & directives from FormatComm and process	CMD: Fop	CMD: Format	once per command
Provide Configur-ation Info	FoGnCmd Fop RmsIF	Receive directives (other than commands)	CMD: Fop	RMS: String Manager	> 2 x 5 per pass, or < 280 / day
Provides access to data values	FoGn Parameter Server	Distribution of updated values to other processes	Parameter Server	CMD: Fop	> 2 x 5 per pass, or < 280 / day
Event Logging	FdEvEvent Logger	Provides routing and archiving of events messages	DMS: FdEvEvent Archiver	CMD: Fop	Once per command

3.3.3 FopCommand Object Model Description

The FopCommand process is an implementation of CCSDS Frame Operation Procedure (FOP) protocol. Together with on board Frame Acceptance and Reporting Mechanism (FARM), it ensures type-A frames to be accepted by the spacecraft only if they are in strict sequential order. It utilizes sequential (" go-back-n") retransmission techniques to correct Telecommand Frames that were rejected by the spacecraft because of error. This process accepts commands (in 1553B format) or memory load packets (in CCSDS packet format) from FormatCommand process, builds them into CCSDS transfer frame format and sends them to TransmitCommand process. The FopCommand process must also accept directives from other processes, as well as process CLCWs from the satellite which keep it up to date on what commands have been accepted onboard.

FcCmCcsdsFop class is the controller of the process. It establishes connections with other subsystems and command processes when initialized. It then waits on system interfaces for inputs from other subsystems and command processes. When an input arrives, FcCmCcsdsFop class asks the corresponding interface class to handle input message. The interface class returns an instance of FoFopRequest. Because the Execute operation of FoFopRequest class is polymorphic, each instance of class derived from FoFopRequest knows which operations of FcCmFopState class need to be invoked to correctly process the specific request. FcCmCcsdsFop then delegates the request to the current active FcCmFopState. A request is processed differently depending on the current active state of the Fop protocol.

FcCmFopState represents the state of Fop protocol. It has six derived classes with each of them represents a different operational Fop state. FcCmFopState class declares an interface common to all of its six subclasses. It also defines all the common behavior of the subsequent derived states. A FcCmFopState object can be in one of six different states: FcCmFopActive, FcCmFopInitial, FcCmInitializeWithBc, FcCmFopInitializeWithoutBc, FcCmFopRxmitWithWait, and FcCmFopRxmitWithoutWait. When FcCmFopState receives a request from FcCmCcsdsFop, it responds differently depending on its current state. For example, the effect of an uplink a new command request depends on whether the FcCmFopState is in its active or initial state.

FcCmFopInitial is a subclass of FcCmFopState. It implements Fop initial state specific behavior.

FcCmFopActive is a subclass of FcCmFopState. It implements Fop active state specific behavior.

FcCmFopInitializeWithoutBc is a subclass of FcCmFopState. It implements Fop "Initializing without BC Frame" state specific behavior.

FcCmFopInitializeWithBc is a subclass of FcCmFopState. It implements Fop "Initializing with BC Frame" state specific behavior.

FcCmFopRmitWithWait is a subclass of FcCmFopState. It implements Fop "Retransmit with Wait" state specific behavior.

FcCmFopRmitWithoutWait is a subclass of FcCmFopState. It implements Fop "Retransmit without Wait" state specific behavior.

FoPsClientIF is the proxy class for parameter server. FopCommand process uses this class to send parameter updates to parameter server.

FoGnCmdFopTransmitProxy is a proxy class provided by TransmitCommand process. It is used by FopCommand process to send CLTUs to TransmitCommand process.

FcCdFopFormatProxy is a proxy class provided by FormatCommand process. It is used by FopCommand process to send acceptance and uplink status to FormatCommand process.

FoGnCmdFopFormatIF is the interface class, from which FopCommand process receives command data from FormatCommand process.

FoGnCmdFopRmsIF is the class from which FopCommand process receives RMS directives.

FoGnCmdFopRmsProxy is the proxy class that FopCommand process provided for RMS subsystem. This class resides in RMS subsystem and is used by RMS to send directives to FopCommand process.

FoGnCmdFopGroundStationIF is the class from which FopCommand process receives CLCW from EDOS.

FoFopRequest is an abstract class. It is the base class for all request objects. When the first input arrives, the corresponding interface interprets the input and creates an instance of a specific request object. At instantiation time, the request object is given the visibility to the FcCmCcsdsFop class, therefor it knows how to invoke the corresponding operation of FcCmCcsdsFop. FcCmRequest has an abstract Execute operation. This allows all request to be executed the same way without having to identify the request.

FoGnRmsReq is a type of FoFopRequest. It defines a common interface for all request from RMS.

FoGnStartAdWithoutClcwReq is a type of FoGnRmsReq. It defines a binding between RMS' "Start AD service without CLCW check" request and StartAdWithoutClcw function of FcCmCcsdsFop class.

FoGnStartAdWithClcwCheckReq is a type of FoGnRmsReq. It defines a binding between RMS's "Start AD service with a CLCW check" request and the StartAdWithClcwChek function of FcCmCcsdsFop class.

FoGnResumeAdServiceReq is a type of FoGnRmsReq. It defines a binding between RMS's "Resume AD service" request and the ResumeAdService function of FcCmCcsdsFop class.

FoGnTerminateAdReq is a type of FoGnRmsReq. It defines a binding between RMS "Terminate AD service" request and TerminateAdService function of FcCmCcsdsFop class.

FoGnSetVsReq is a type of FoGnRmsReq. It defines a binding between RMS's "Set ground transmitter sequence number" request and the SetVs function of FcCmCcsdsFop class.

FoGnSetWinWidthReq is a type of FoGnRmsReq. It defines a binding between RMS's "Set Fop sliding window width" request and the SetWinWidth function of FcCmCcsdsFop class.

FoGnSetTimeInitialValReq is a type of FoGnRmsReq. It defines a binding between RMS's "Set timeout initial value" request and SetTimeInitialVal function of FcCmCcsdsFop class.

FoGnSetTransmissionLimitReq is a type of FoGnRmsReq. It defines a binding between RMS's "Set transmission Limit" request and the SetTransmissionLimit function of FcCmCcsdsFop class.

FoGnSelectCtiuReq is a type of FoGnRmsReq. It defines a binding between RMS's " Select Ctiu" request and the SelectCtiu function of FcCmCcsdsFop class.

FoGnGetConfigSnapshotReq is a type of FoGnRmsReq. It defines a binding between RMS "Get Configuration snapshot" request and GetConfigSnapshot function of FcCmCcsdsFop class.

FoGnChangeRoleReq is a type of FoGnRmsReq. It defines a binding between RMS "change the

state of current string (to Primary, Backup or Inactive)" request and the ChangeRole function of FcCmCcsdsFop class.

FoGnShutdownFopReq is a type of FoGnRmsReq. It defines a binding between RMS "shut down Fop" request and the ShutdownFop function of FcCmCcsdsFop class.

FcGnFormatProcessReq is a type of FoFopRequest. It defines a common interface for all request from FormatComand process.

FcGnProcessRtCmdReq is a type of FcGnFormatProcessReq. It defines a binding between a request to uplink a real time command (in 1553B format) and the ProcessRtCmd function of FcCmCcsdsFop class.

FcGnProcessLoadPacketReq is a type of FcGnFormatCommandReq. It defines a binding between a request to uplink a memory load packet (in CCSDS packet format) and the ProcessLoadPacket function of the FcCmCcsdsFop class.

FoGnProcessClcwReq is a type of FoFopRequest. This class is responsible for preliminary processing of a CLCW. It first checks the validity of a CLCW bit pattern against the CCSDS standard. It then decommutates the CLCW. This request also defines a binding between a CLCW and the ProcessClcw function of FcCmCcsdsFop class.

FcCmTcFrame is responsible for building transfer frame according to the CCSDS format. When receives a real time command (in 1553B format) or a memory load packet (in CCSDS packet format) from FcCmCcsdsFop class, FcCmFopState first creates an instance of FcCmTcFrame which knows how to build a CCSDS frame from a 1553B real time command or a memory load packet. FcCmFopState then asks FcCmTcFrame class to build a CCSDS frame and also the corresponding CLTU. Once this is done, FcCmFopState sends the CLTU to the TransmitCommand process via its proxy where the CLTU is uplinked. Finally, FcCmFopState saves the current instance of FcCmTcFrame in the command sent queue until the frame is receipt verified by a CLCW. Once this command is CLCW verified, the copy of FcCmTcFrame is then deleted from the command sent queue. FcCmFcFrame is the aggregation of FcCmTcFrameHeader, FcCmFrameData and FcCmFrameCrc. Each part knows how to build itself on demand.

FcCmTcFrameHeader is responsible for building transfer frame header according to the CCSDS format.

FcCmTcFrameData is responsible for building transfer frame data portion. It is the aggregation of FcCmTcPacketHeader and FcCmTcPacketData. Each part knows how to build itself on demand.

FcCmFcFrameCrc is responsible for calculating the crc check code for the entire content of a transfer frame.

FcCmTcPacketHeader is responsible for building the Tc Packet header according to the CCSDS format.

FcCmTcPacketData is responsible for building the packet data portion according to the CCSDS format.

FcGnTcCltu is the data structure to be send to TransmitCommand process. After the frame is build, FcCmTcFrame creates an instance of FcGnTcCltu which contains the CLTU for current transfer frame. The FcGnTcCltu object is then passed to Command Transmit process. It is then uplinked to satellite via EDOS.

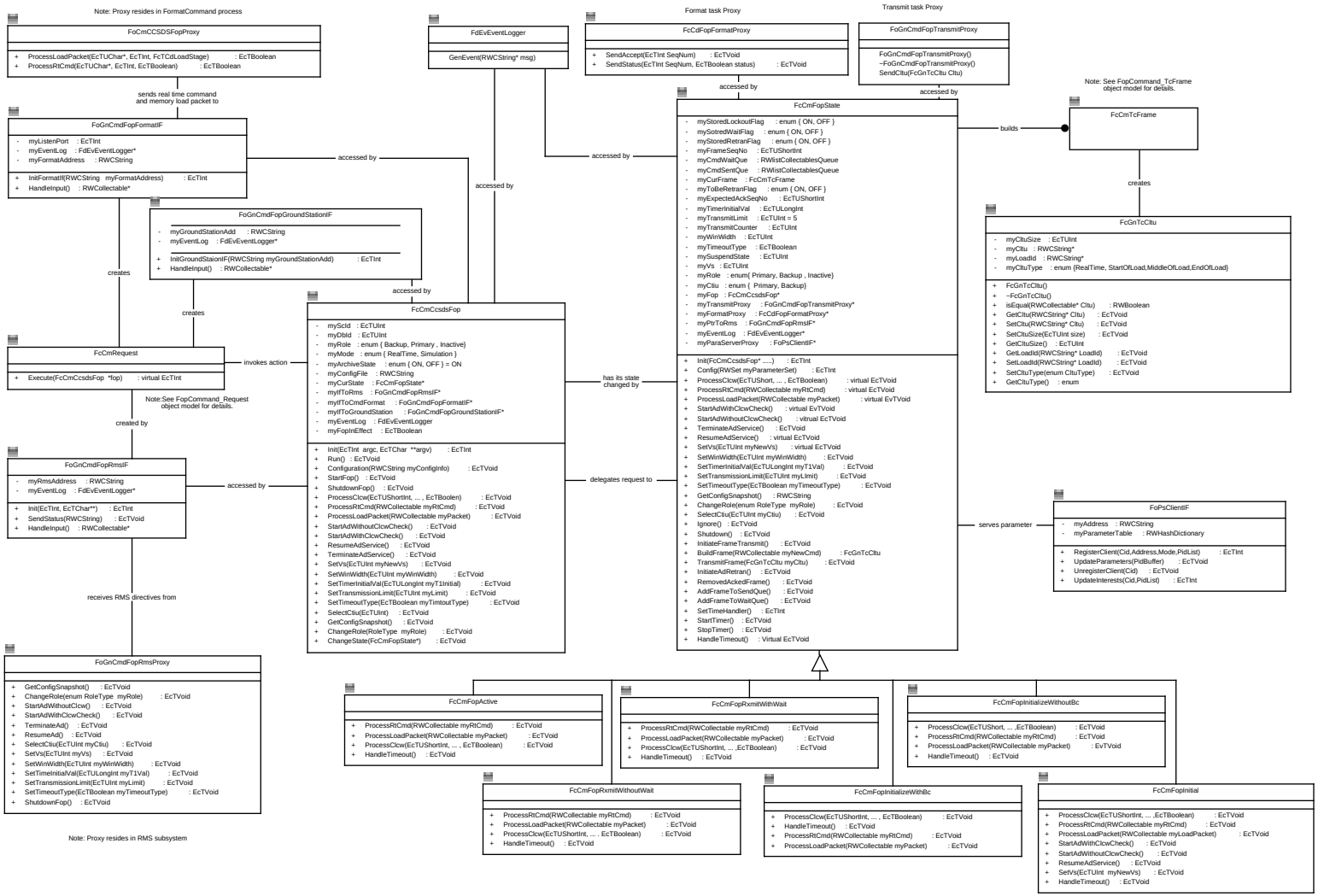

Figure 3.3.3-1. FopCommand Object Diagram

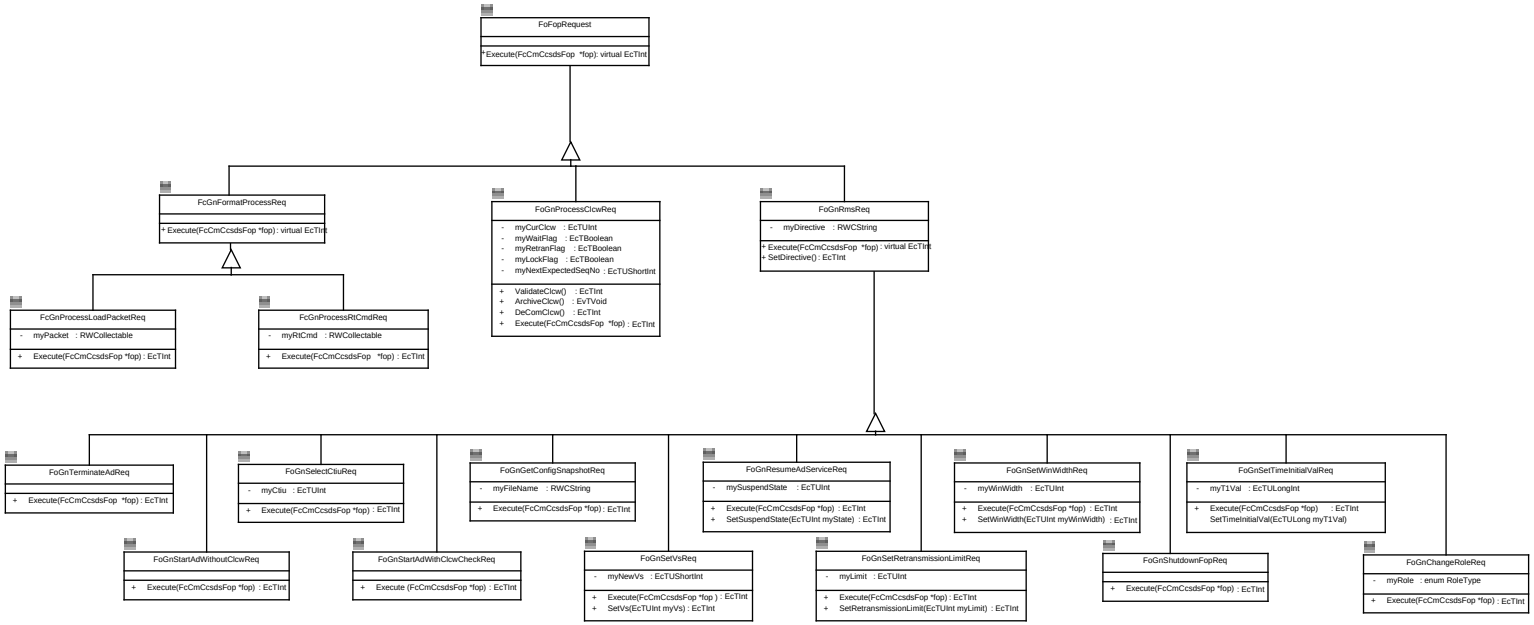

Figure 3.3.3-2. FopCommand Request Message Object Diagram

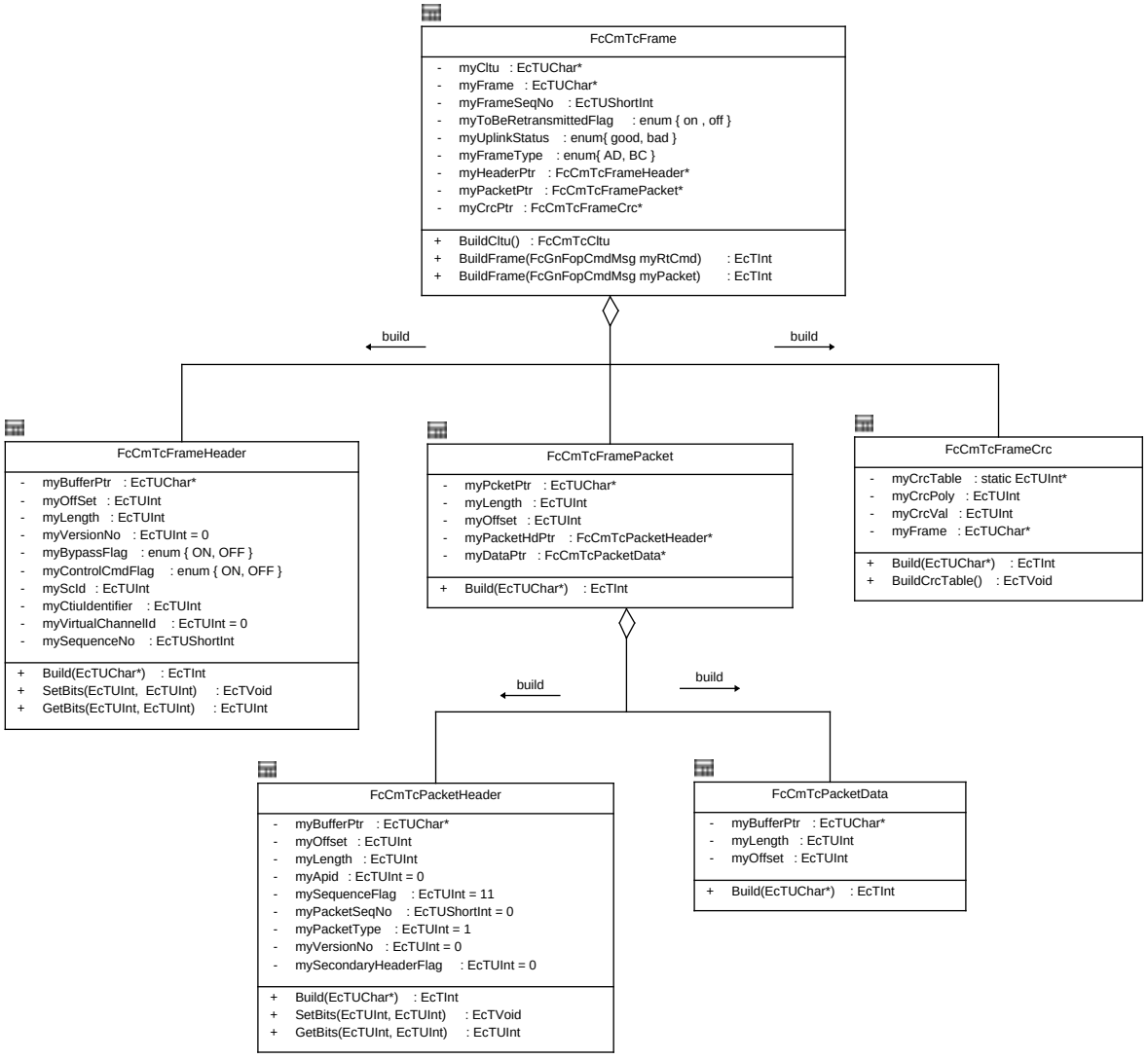

Figure 3.3.3-3. FopCommand TcFrame Object Diagram

3.3.4 FopCommand Dynamic Model Description

The following are the FopCommand scenarios which are defined in this section.

Real-Time Command FOP Initialization: Successful

Real-Time Command FOP Initialization: Failure

Real-Time Command FOP Init. AD Service w/out CLCW: Successful

Real-Time Command FOP Init. AD Service w/out CLCW: Failure

Real-Time Command FOP Init. AD Service with CLCW: Successful

Real-Time Command FOP Init. AD Service with CLCW: Failure

Real-Time Command FOP Init. AD Service with Set VR: Successful

Real-Time Command FOP Init. AD Service with Set VR: Failure

Real-Time Command FOP Command Transmission

Real-Time Command FOP Command Retransmission

Additionally, a state diagram for the FcCmCcsdsFop object is included.

3.3.4.1 Real-Time Command FOP Initialization: Successful Scenario

3.3.4.1.1 Real-Time Command FOP Initialization: Successful Abstract

The purpose of the "Real-Time Command FOP Initialization: Successful" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process is initialized.

Figure 3.3.4.1-1 is the event trace diagram which corresponds to this scenario.

3.3.4.1.2 Real-Time Command FOP Initialization: Successful Summary Information

Interfaces:

Parameter Server

Data Management Subsystem

Resource Management Subsystem

FormatCommand

TransmitCommand

Stimulus:

The Resource Manager (RMS) starts up the FopCommand process.

Desired Response:

The Resource Management receives the status of successful FOP initialization.

Pre-Conditions:

Configuration file must be identified and available.

Post-Conditions:

The FOP is placed in the "initial" state, and ready for directives.

3.3.4.1.3 Scenario Description

The main operation of the FopCommand application (FcCmFopAppl) is invoked when the Resource Manager (RMS) starts up the process. The command line will contain the IP address of the RMS. This address is forwarded to the FcCmCcsdsFop, the controller of the FOP processing. The IP address is used to establish communication with the RMS, via FoGnCmdFopRmsIF. Once communication is established, FopCommand process sends a message to RMS subsystem and informs it that the process is ready. The FopCommand then waits for a configuration request message from RMS. Upon receipt of the message, the message is read and input parameters are extracted from the message. These parameters contain IP addresses which are used to establish communications with other processes, specifically Parameter Server, and the Command processes FormatCommand and TransmitCommand. Other parameters include the spacecraft ID, database ID, and the process "role" as part of a either a primary or backup string. The DMS and EDOS addresses are looked up from a name server.

A DMS connection is established via FdEvEventLogger for events processing.

FoDsFile is then utilized to access the database file. The file information is used to configure the FOP attributes and will contain default values for various attributes. This configuration information is then used to configure the FOP.

Next, the Parameter Server, the FormatCommand process and the TransmitCommand process connections are established via FoPsClientIF, FcCmdFopFormatIF and FcCmFopTransmitProxy objects respectively.

Finally, an FcCmFopState object (as a derived FcCmFopInitial object) is created and initialized, and the initialization is complete.

A "successful initialization" event message is logged via FdEvEventLogger, and a successful completion status is returned to RMS.

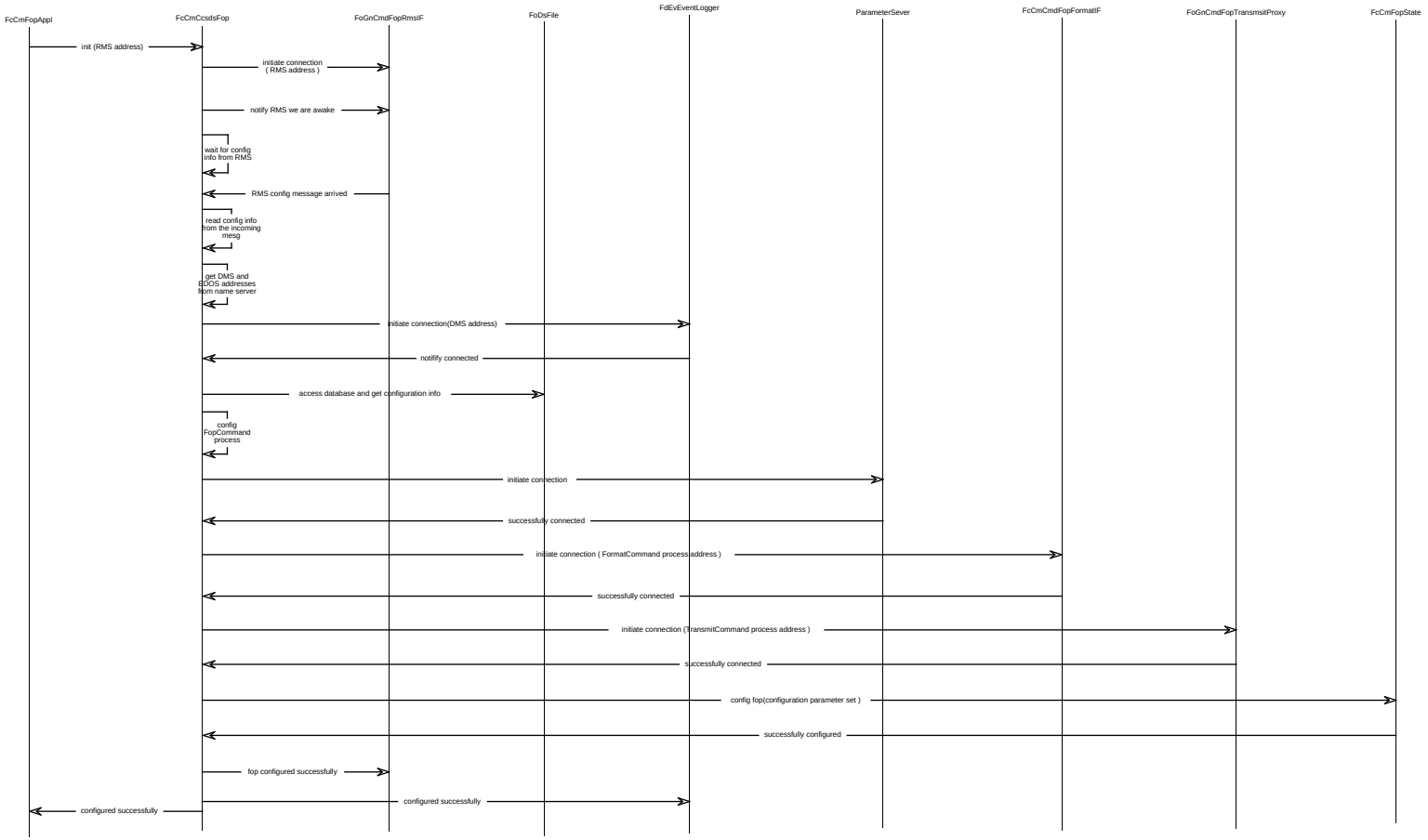

Figure 3.3.4.1-1. FopCommand Initialization: Successful

3.3.4.2 Real-Time Command FOP Initialization: Failure Scenario

3.3.4.2.1 Real-Time Command FOP Initialization: Failure Abstract

The purpose of the "Real-Time Command FOP Initialization: Failure" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process handles a fatal error during initialization.

Figure 3.3.4.2-1 is the event trace diagram which corresponds to this scenario.

3.3.4.2.2 Real-Time Command FOP Initialization: Failure Summary Information

Interfaces:

Data Management Subsystem

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) starts up the FopCommand process.

Desired Response:

The Resource Management receives the failure status regarding FOP initialization.

Pre-Conditions:

Configuration file must be identified and available.

Post-Conditions:

The Resource Management is notified of the failure.

3.3.4.2.3 Scenario Description

The main operation of the FopCommand application (FcCmFopAppl) is invoked when the Resource Manager (RMS) starts up the process. The command line will contain the IP address of the RMS. This address is forwarded to the FcCmCcsdsFop, the controller of the FOP processing. The IP address is used to establish communication with the RMS, via FoGnCmdFopRmsIF. Once communication is established, FopCommand process sends a message to RMS subsystem and informs it that the process is ready. The FopCommand then waits for a configuration request message from RMS. Upon receipt of the message, the message is read and input parameters are extracted from the message. These parameters contain IP addresses which are used to establish communications with other processes, specifically the Parameter Server, and the Command processes FormatCommand and TransmitCommand. Other parameters include the spacecraft ID, database ID, and the process "role" as part of a either a primary or backup string. The DMS and EDOS addresses are looked up from a name server.

An attempt is made to establish connection with DMS via FdEvEventLogger for events processing. However, the connection is unsuccessful, and a Failure completion status is returned to RMS.

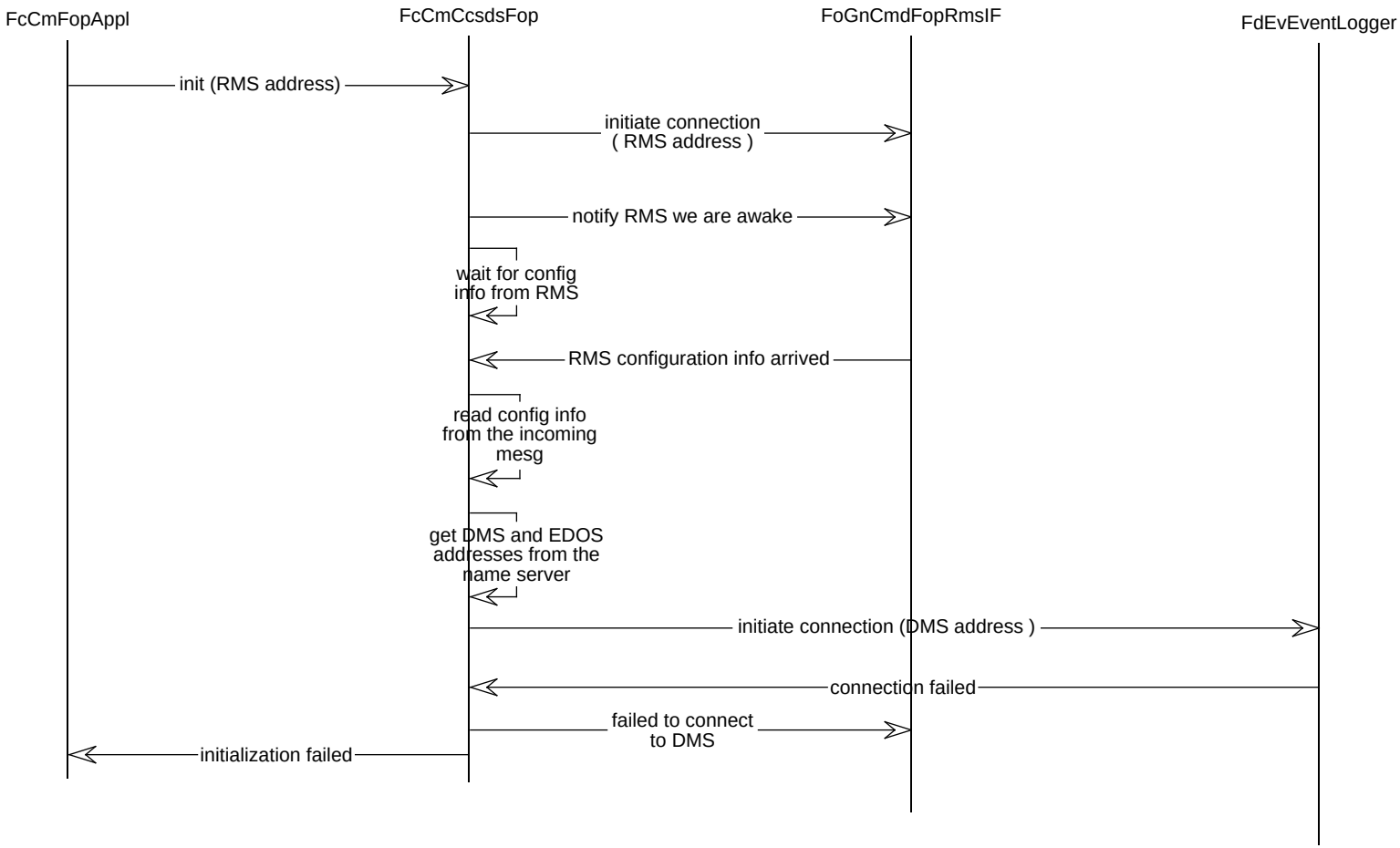

Figure 3.3.4.2-1. FopCommand Initialization: Failure Scenario

3.3.4.3 Real-Time Command FOP Init. AD Service w/out CLCW: Successful Scenario

3.3.4.3.1 Real-Time Command FOP Init. AD Service w/out CLCW: Successful Abstract

The purpose of the "Real-Time Command FOP Init. AD Service w/out CLCW: Successful" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand task is configured to uplink commands, without waiting for a "clear" CLCW.

Figure 3.3.4.3-1 is the event trace diagram which corresponds to this scenario.

3.3.4.3.2 Real-Time Command FOP Init. AD Service w/out CLCW: Successful Summary Information

Interfaces:

Data Management Subsystem

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) forwards the "Initialize w/out CLCW" directive.

Desired Response:

The FOP is placed in the "active" state.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

The FOP is in "active" state and ready to process formatted commands.

3.3.4.3.3 Scenario Description

The FOP is in the "initial" state and waiting for directives. Upon arrival of the "initialize w/out clcw" directive from RMS, the FoGnCmdFopRmsIF creates a request object which corresponds to the directive; FoGnStartAdWithoutClcwReq. This object, in turn, echoes the directive via FoEvEventLogger. It then indirectly (via FcCmCcsdsFop) invokes the StartAdWithoutClcwCheck operation of FcCmFopState, which successfully initializes the FOP. A success event message is logged, the Resource Manager is notified of the successful processing of the directive and the FOP is placed in the "active" state.

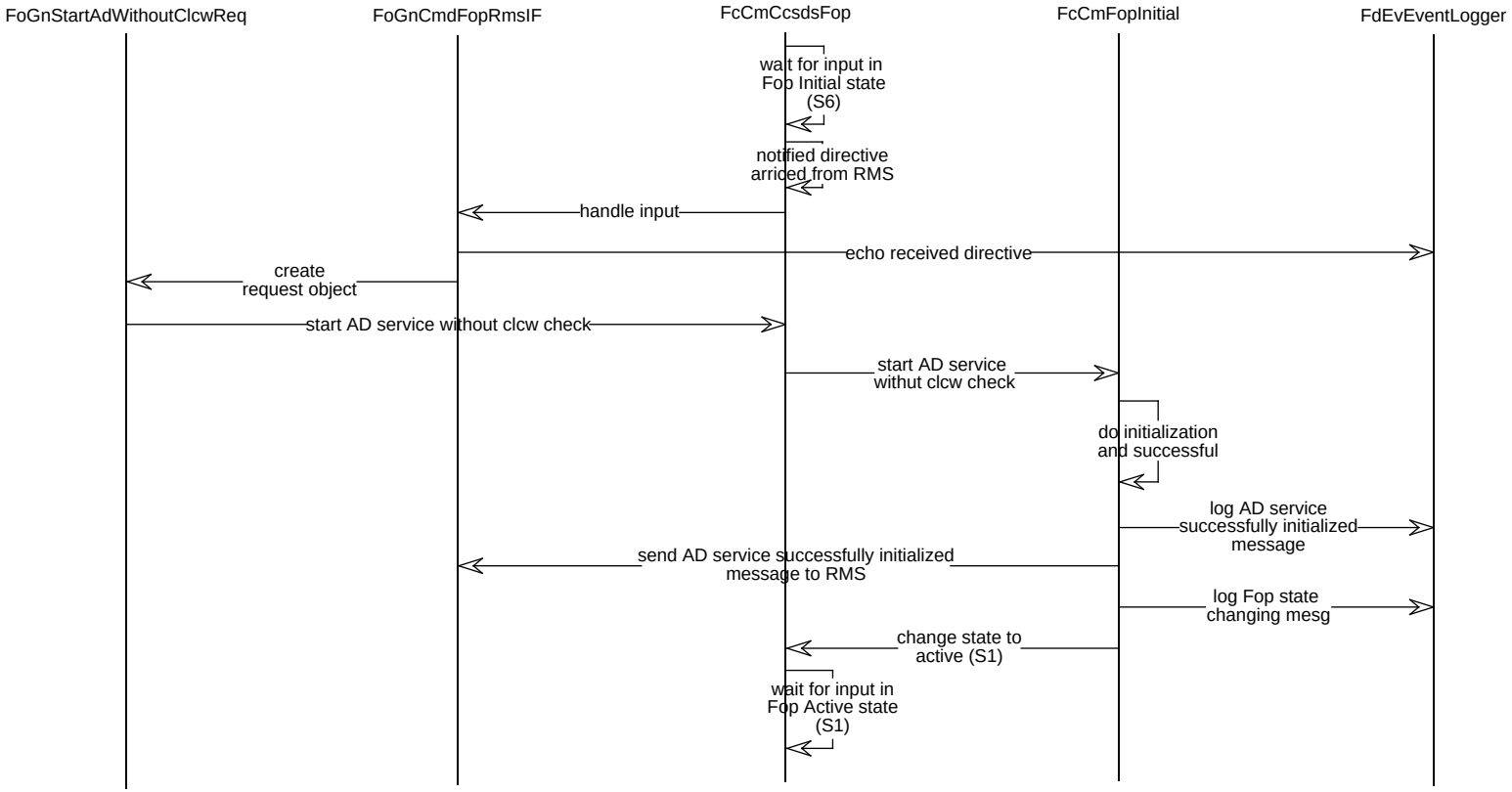

Figure 3.3.4.3-1. FopCommand Init. AD Service w/out CLCW: Successful

3.3.4.4 Real-Time Command FOP Init. AD Service w/out CLCW: Failure Scenario

3.3.4.4.1 Real-Time Command FOP Init. AD Service w/out CLCW: Failure Abstract

The purpose of the "Real-Time Command FOP Init. AD Service w/out CLCW: Failure" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand task recovers from an unsuccessful attempt to configured for uplink commands, without waiting for a "clear" CLCW.

Figure 3.3.4.4-1 is the event trace diagram which corresponds to this scenario.

3.3.4.4.2 Real-Time Command FOP Init. AD Service w/out CLCW: Failure Summary Information

Interfaces:

Data Management Subsystem

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) forwards the "Initialize w/out CLCW" directive.

Desired Response:

The FOP remains in the "initial" state.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

FOP is in the "initial" state.

3.3.4.4.3 Scenario Description

The FOP is in the "initial" state and waiting for directives. Upon arrival of the "initialize w/out clcw" directive from RMS, the FoGnCmdFopRmsIF creates a request object which corresponds to the directive; FoGnStartAdWithoutClcwReq. This object, in turn, echoes the directive via FdEvEventLogger. It then indirectly (via FcCmCcsdsFop) invokes the StartAdWithoutClcwCheck operation of FcCmFopState, which fails to initialize the FOP. A failure event message is logged, the Resource Manager is notified of the failure to process the directive and the FOP is left in the "initial" state.

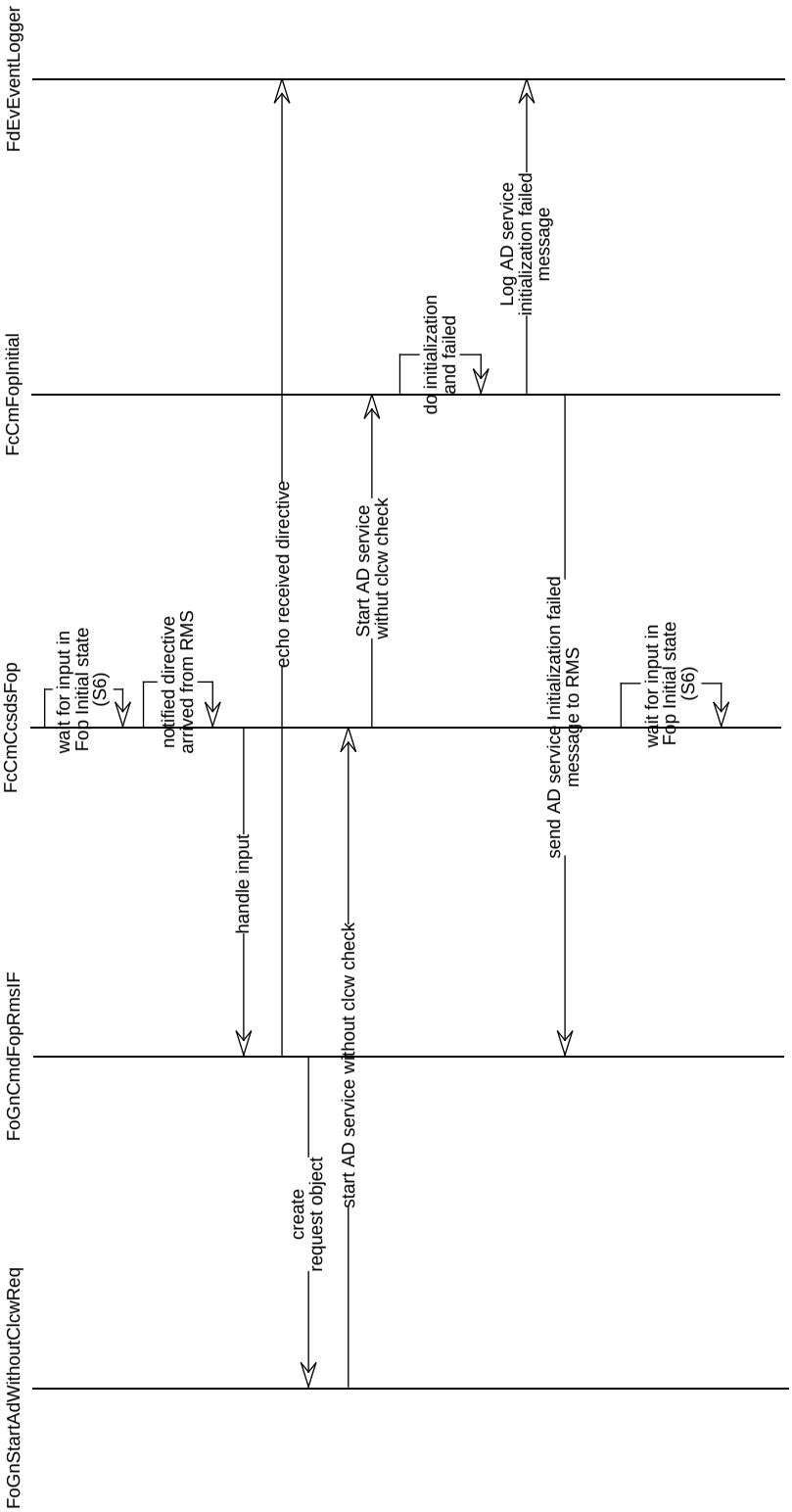

Figure 3.3.4.4-1. FopCommand Init. AD Service w/out CLCW: Failure scenario

3.3.4.5 Real-Time Command FOP Init. AD Service with CLCW: Successful Scenario

3.3.4.5.1 Real-Time Command FOP Init. AD Service with CLCW: Successful Abstract

The purpose of the "Real-Time Command FOP Init. AD Service with CLCW: Successful" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process is configured to uplink commands upon receipt of a "clean" CLCW; i.e., the Wait, Retransmit and Lockout flags are "off", and the sequence number equals its expected value.

Figure 3.3.4.5-1 is the event trace diagram which corresponds to this scenario.

3.3.4.5.2 Real-Time Command FOP Init. AD Service with CLCW: Successful Summary Information

Interfaces:

EDOS

Data Management Subsystem

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) forwards the "Initialize w/out CLCW" directive.

Desired Response:

The FOP is placed in the "active" state.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

The FOP is in "active" state and ready to process formatted commands.

3.3.4.5.3 Scenario Description

The FOP is in the "initial" state and waiting for directives. Upon arrival of the "initialize with CLCW" directive from RMS, the FoGnCmdFopRmsIF creates a request object that corresponds to the directive; FoGnStartAdWithClcwReq. It then indirectly (via FcCmCcsdsFop) invokes the StartAdWithClcwCheck operation of FcCmFopState, which successfully initializes the FOP and sets the timer (used for detection of certain retransmission circumstances). An event message for state change is issued, and FopProcess goes into a wait, in the "initializing without BC frame" state, waiting for either the timer, or a directive.

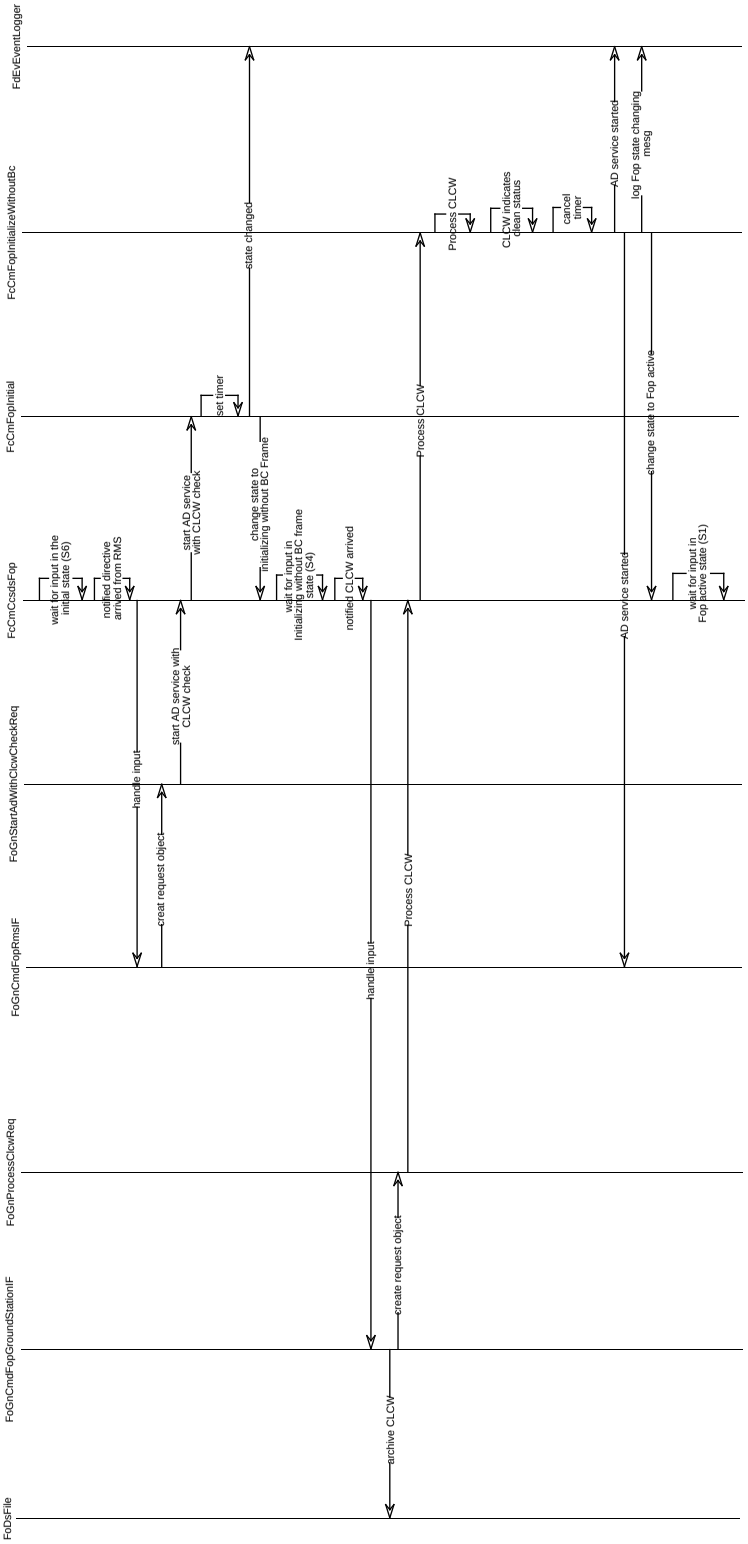

Figure 3.3.4.5-1. FopCommand Init. AD Service with CLCW: Successful

Upon arrival of a CLCW, the FoGnCmdFopGroundStationIF archives the CLCW via FoDsFile, and creates a request object that corresponds to the directive; FoGnProcessClcwReq. FoGnProcessClcwReq routes the CLCW through FcCmCcsdsFop to FcCmFopInitializeWithoutBc for processing. The CLCW indicates a "clean" status, so a success event message is logged, the Resource Manager is notified of the successful processing of the directive and the FOP is placed in the "active" state.

3.3.4.6 Real-Time Command FOP Init. AD Service with CLCW: Failure Scenario

3.3.4.6.1 Real-Time Command FOP Init. AD Service with CLCW: Failure Abstract

The purpose of the "Real-Time Command FOP Init. AD Service with CLCW: Failure" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process responds to a failure in the attempt to configure to uplink commands upon receipt of a "clean" CLCW.

Figure 3.3.4.6-1 is the event trace diagram which corresponds to this scenario.

3.3.4.6.2 Real-Time Command FOP Init. AD Service with CLCW: Failure Summary Information

Interfaces:

EDOS

Data Management Subsystem

Resource Management Subsystem

Stimulus:

The Resource Manager (RMS) forwards the "Initialize w/out CLCW" directive.

Desired Response:

Appropriate error messages are logged, and the FOP remains in the "initial" state.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

The FOP remains in "initial" state.

3.3.4.6.3 Scenario Description

The FOP is in the "initial" state and waiting for directives. Upon arrival of the "initialize with CLCW" directive from RMS, the FoGnCmdFopRmsIF creates a request object that corresponds to the directive; FoGnStartAdWithClcwReq. It then indirectly (via FcCmCcsdsFop) invokes the StartAdWithClcwCheck operation of FcCmFopState, which successfully initializes the FOP and sets the timer (used for detection of certain retransmission circumstances). An event message for state change is issued, and FopCommand process goes into a wait, in the "initializing without BC frame" state, waiting for either the timer, or a directive.

Upon arrival of a CLCW, the FoGnCmdFopGroundStationIF archives the CLCW via FoDsFile, and creates a request object that corresponds to the directive; FoGnProcessClcwReq. FoGnProcessClcwReq routes the CLCW through FcCmCcsdsFop to FcCmFopInitializeWithoutBc for processing. The CLCW, however, indicates a "not clean" status so a failure event message is logged, the Resource Manager is notified of the failure in the processing of the directive, and the FOP remains in the "initial" state.

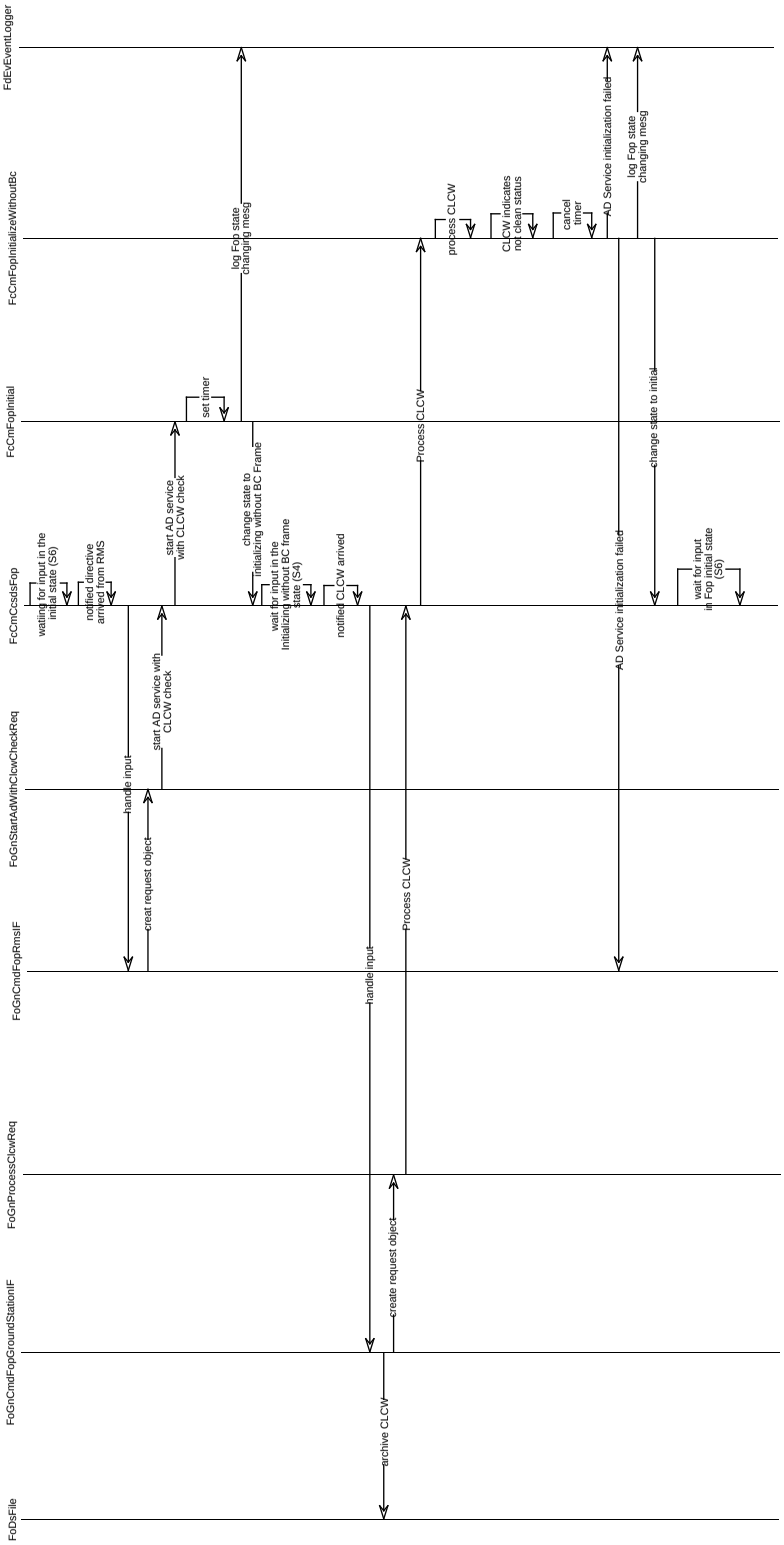

Figure 3.3.4.6-1. FopCommand Init. AD Service with CLCW: Failure scenario

3.3.4.7 Real-Time Command FOP Init. AD Service with Set VR: Successful Scenario

3.3.4.7.1 Real-Time Command FOP Init. AD Service with Set VR: Successful Abstract

The purpose of the "Real-Time Command FOP Init. AD Service with Set VR: Successful" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process is configured to uplink commands in the "set VR" mode. This includes uplinking of the BC type frame, which sets the FARM's sequence counter (VR) onboard the spacecraft.

Figure 3.3.4.7-1 is the event trace diagram which corresponds to this scenario.

3.3.4.7.2 Real-Time Command FOP Init. AD Service with Set VR: Successful Summary Information

Interfaces:

EDOS

Data Management Subsystem

FormatCommand

TransmitCommand

Stimulus:

FormatCommand forwards the "Initialize with set VR" command.

Desired Response:

The FOP is placed in the "active" state, and the sequence counter onboard the spacecraft has been set to the specified value.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

The FOP is in "active" state and ready to process formatted commands.

3.3.4.7.3 Scenario Description

The FOP is in the "initial" state and waiting for inputs. Upon arrival of a command from CommandFormat process, the controller directs FoGnCmdFopFormatIF to create a request object that corresponds to the incoming command, FcGnProcessRtCmdReq. It then indirectly (via FcCmCcsdsFop) invokes the ProcessRtCmd operation of FcCmFopInitial, which realizes it is a control command by checking the command type flag. The FcCmFopInitial then prepares CLTU of a BC transfer frame for uplinking.

The Transfer Frame is composed of a CCSDS Packet, so this is first built (via FcCmTcPacketData and FcCmTcPacketHeader). The Transfer Frame is completed by adding a header (via FcCmTcFrameHeader) and calculating a CRC (via FcCmTcFrameCrc).

The completed transfer frame is then further processed into a CLTU (via FcCmTcCltu) which is then added to the queue of commands sent, and forwarded to the TransmitCommand process (via FoGnCmdFopTransmitProxy).

The timer (used for detection of certain retransmission circumstances) is then set, and control is

returned to FcCmCcsdsFop in the Initializing with BC frame state, and waits for a CLCW confirming its receipt.

Upon arrival of a CLCW, the FoGnCmdFopGroundStationIF archives the CLCW via FoDsFile, and creates a request object that corresponds to the directive; FoGnProcessClcwReq. FoGnProcessClcwReq routes the CLCW through FcCmCcsdsFop to FcCmFopInitializeWithBc for processing. The CLCW indicates a "clean" status, so a success event message is logged, the Resource Manager is notified of the successful processing of the directive and the FOP is placed in the "active" state.

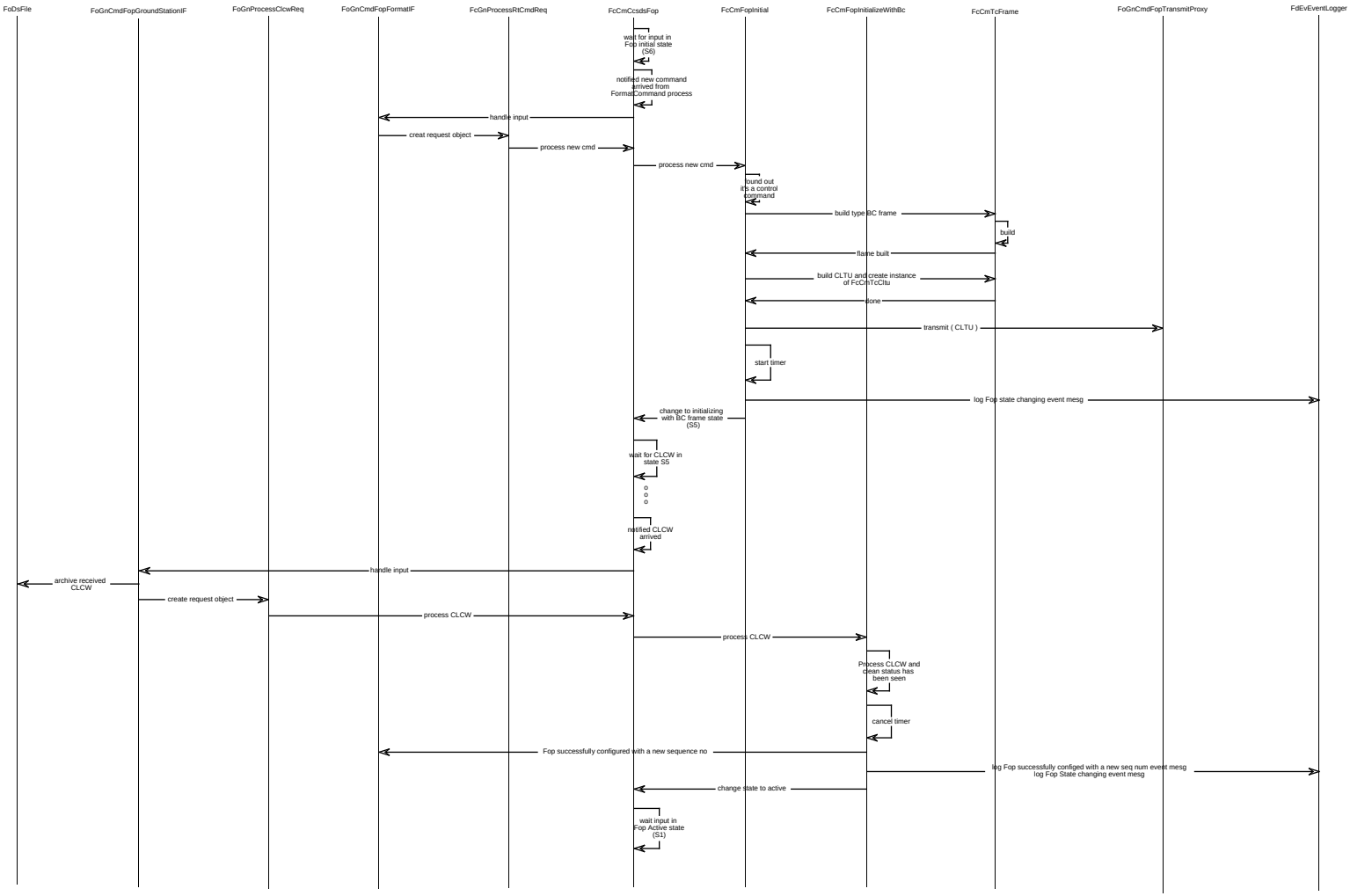

Figure 3.3.4.7-1. FopCommand Init. AD Service with set VR: Successful scenario

3.3.4.8 Real-Time Command FOP Init. AD Service with Set VR: Failure Scenario

3.3.4.8.1 Real-Time Command FOP Init. AD Service with Set VR: Failure Abstract

The purpose of the "Real-Time Command FOP Init. AD Service with Set VR: Failure" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process responds to a failure in the attempt to configured to uplink commands in the "set VR" mode.

Figure 3.3.4.8-1 is the event trace diagram which corresponds to this scenario.

3.3.4.8.2 Real-Time Command FOP Init. AD Service with Set VR: Failure Summary Information

Interfaces:

EDOS

Data Management Subsystem

FormatCommand

TransmitCommand

Stimulus:

FormatCommand forwards the "Initialize with set VR" command.

Desired Response:

Appropriate error messages are logged, and the FOP remains in the "initial" state.

Pre-Conditions:

FOP is in the "initial" state.

Post-Conditions:

The FOP remains in "initial" state.

3.3.4.8.3 Scenario Description

The FOP is in the "initial" state and waiting for inputs. Upon arrival of a command from CommandFormat process, the controller directs FoGnCmdFopFormatIF to create a request object that corresponds to the incoming command, FcGnProcessRtCmdReq. It then indirectly (via FcCmCcsdsFop) invokes the ProcessRtCmd operation of FcCmFopInitial, which realizes it is a control command by checking the command type flag. The FcCmFopInitial then prepares CLTU of a BC transfer frame for uplinking.

The Transfer Frame is composed of a CCSDS Packet, so this is first built (via FcCmTcPacketData and FcCmTcPacketHeader). The Transfer Frame is completed by adding a header (via FcCmTcFrameHeader) and calculating a CRC (via FcCmTcFrameCrc).

The completed transfer frame is then further processed into a CLTU (via FcCmTcCltu) which is then added to the queue of commands sent, and forwarded to the TransmitCommand process (via FoGnCmdFopTransmitProxy).

The timer (used for detection of certain retransmission circumstances) is then set, and control is returned to FcCmCcsdsFop in the Initializing with BC frame state, and waits for a CLCW confirming its receipt.

Upon arrival of a CLCW, the FoGnCmdFopGroundStationIF archives the CLCW via FoDsFile, and creates a request object that corresponds to the directive; FoGnProcessClcwReq. FoGnProcessClcwReq routes the CLCW through FcCmFop to FcCmFopInitializeWithBc for processing. The CLCW indicates a "not clean" status, so the FOP remains in the "Initializing with BC frame" state.

A few more CLCWs are received, but none indicate a "clean" status.

Eventually, the timer expires before a CLCW indicating a "clean" status is received, and the BC frame is retransmitted via FoGnCmdFopTransmitProxy. The timer is reset and the transmit counter is incremented by 1.

The retransmission takes place a few more times, until eventually the transmit counter exceeds the transmit limit. The directive is deemed as having failed at this point. FOP shutdown procedure is commenced, an event message indicating directive failure is logged via FdEvEventLogger, FormatCommand is notified of the failure via FoGnCmdFopFormatIF, and the FOP is returned to the "initial" state.

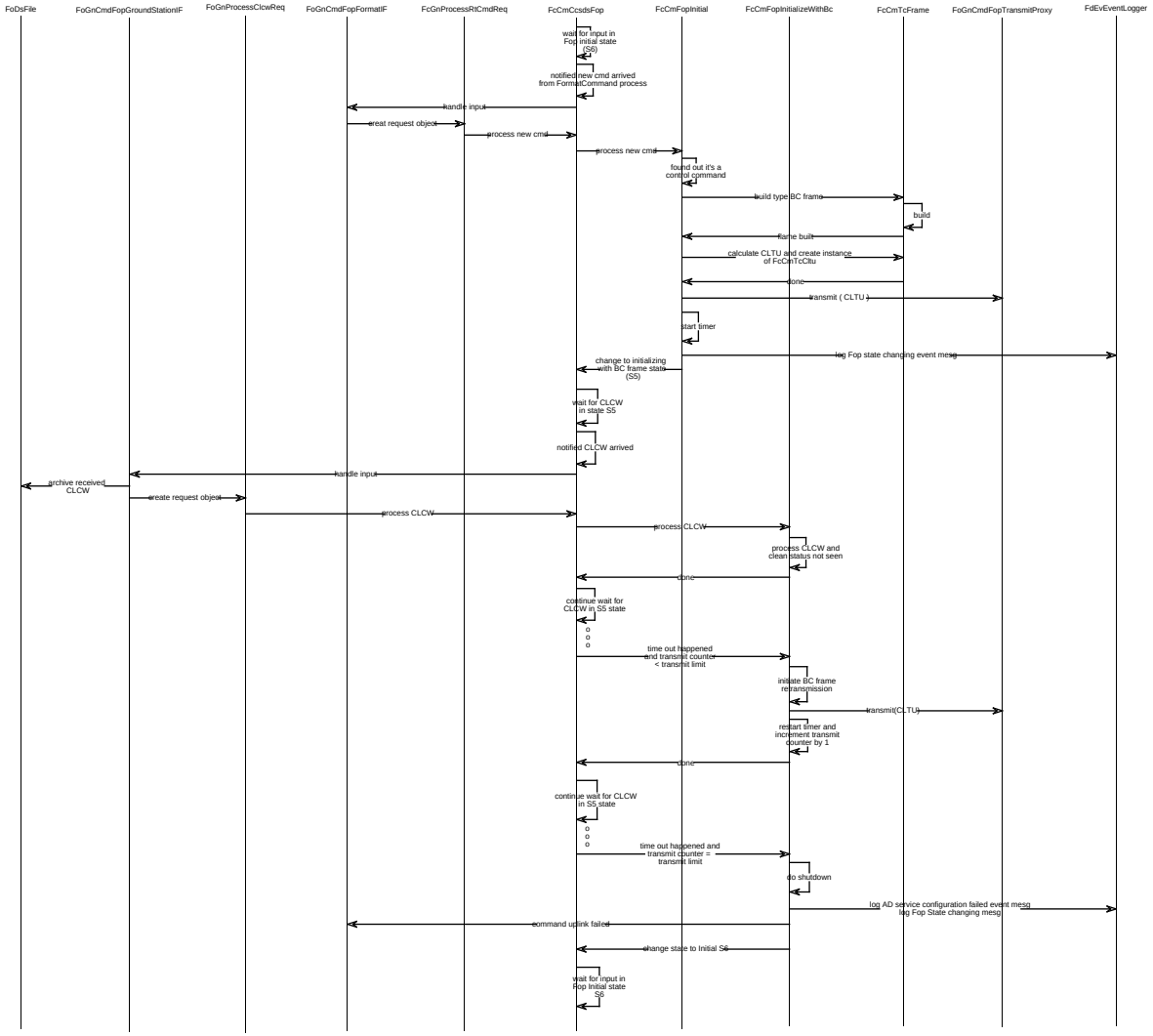

Figure 3.3.4.8-1. FopCommand Init. AD Service with set VR: Failure scenario

3.3.4.9 Real-Time Command FOP Command Transmission Scenario

3.3.4.9.1 Real-Time Command FOP Command Transmission Abstract

The purpose of the "Real-Time Command FOP Command Transmission" scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process processes a 1553-b command received from the FormatCommand process into a CLTU conforming to CCSDS standards for the AM-1 spacecraft, and forwards the CLTU to the TransmitCommand process.

Figure 3.3.4.9-1 and 3.3.4.9-2 are the event trace diagrams which correspond to this scenario.

3.3.4.9.2 Real-Time Command FOP Command Transmission Summary Information

Interfaces:

Data Management Subsystem

EDOS

FormatCommand

TransmitCommand

Stimulus:

A 1553-b command is forwarded to FopCommand by FormatCommand.

Desired Response:

CLTUs are forwarded to TransmitCommand.

Pre-Conditions:

FOP is in the "active" state.

Post-Conditions:

FOP remains in the "active" state.

3.3.4.9.3 Scenario Description

The scenario begins with FopCommand waiting for input in the Active state. FormatCommand process forwards the first of three 1553-b commands in this scenario to FopCommand. The following events transpire in response to receipt of a command, and constitutes the "send" portion of FopCommand processing:

The command is echoed via FdEvEventLogger. FoGnCmdFopFormatIF creates the appropriate object to process the command directive: a FcGnProcessRtCmdReq object.

The Execute operation of this new FcGnProcessRtCmdReq object is invoked, which initiates the building of a CCSDS Transfer Frame (via FcCmTcFrame).

The details for building the Transfer Frame are shown in Figure 3.3.4.9-2. The Transfer Frame is composed of a CCSDS Packet, so this is first built (via FcCmTcPacketData and FcCmTcPacketHeader). The Transfer Frame is completed by adding a header (via FcCmTcFrameHeader) and calculating a CRC (via FcCmTcFrameCrc).

The completed transfer frame is then further processed into a CLTU (via FcCmTcCltu) which is then added to the queue of commands sent, and forwarded to the TransmitCommand process (via

FoGnCmdFopTransmitProxy).

The timer (used for detection of certain retransmission circumstances) is then set, and control is returned to FcCmCcsdsFop in the Active state, thus completing the "send" portion of the FopCommand processing, for the first command in this scenario.

A CLCW arrives, and FoGnCmdGroundStationIF handles it. The CLCW is archived via FoDsFile, and FoGnProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that no new frames are acknowledged onboard the spacecraft. Control is returned to FcCmCcsdsFop in the Active state, thus completes processing of the first CLCW in this scenario.

FormatCommand process forwards the the second 1553-b command in to FopCommand. The "send" portion of FopCommand processing for this command is identical to that for the first command.

FormatCommand process forwards the the third 1553-b command in to FopCommand. Again, the "send" portion of FopCommand processing for this command is also identical to that for the first command.

A second CLCW arrives, and FoGnCmdGroundStationIF handles it. The CLCW is archived via FoDsFile, and FoGnProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that the first command sent in this scenario was received onboard the spacecraft. The CLTU is removed from the queue of commands sent, and an uplink verification event message is issued via FdEvEventLogger. The FormatCommand process is informed of the successful uplink status of the command via FoGnCmdFopFormatIF, and control is returned to FcCmCcsdsFop in the Active state, thus completes processing of the second CLCW in this scenario.

A third CLCW arrives, and FoGnCmdGroundStationIF handles it. The CLCW is archived via FoDsFile, and FcCmProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that both the second and third commands sent in this scenario were received onboard the spacecraft. The two CLTUs are removed from the queue of commands sent, and an uplink verification event message is issued via FdEvEventLogger. The FormatCommand process is informed of the successful uplink status of these commands via FoGnCmdFopFormatIF, and control is returned to FcCmCcsdsFop in the Active state, thus completes processing of the third (and last) CLCW in this scenario.

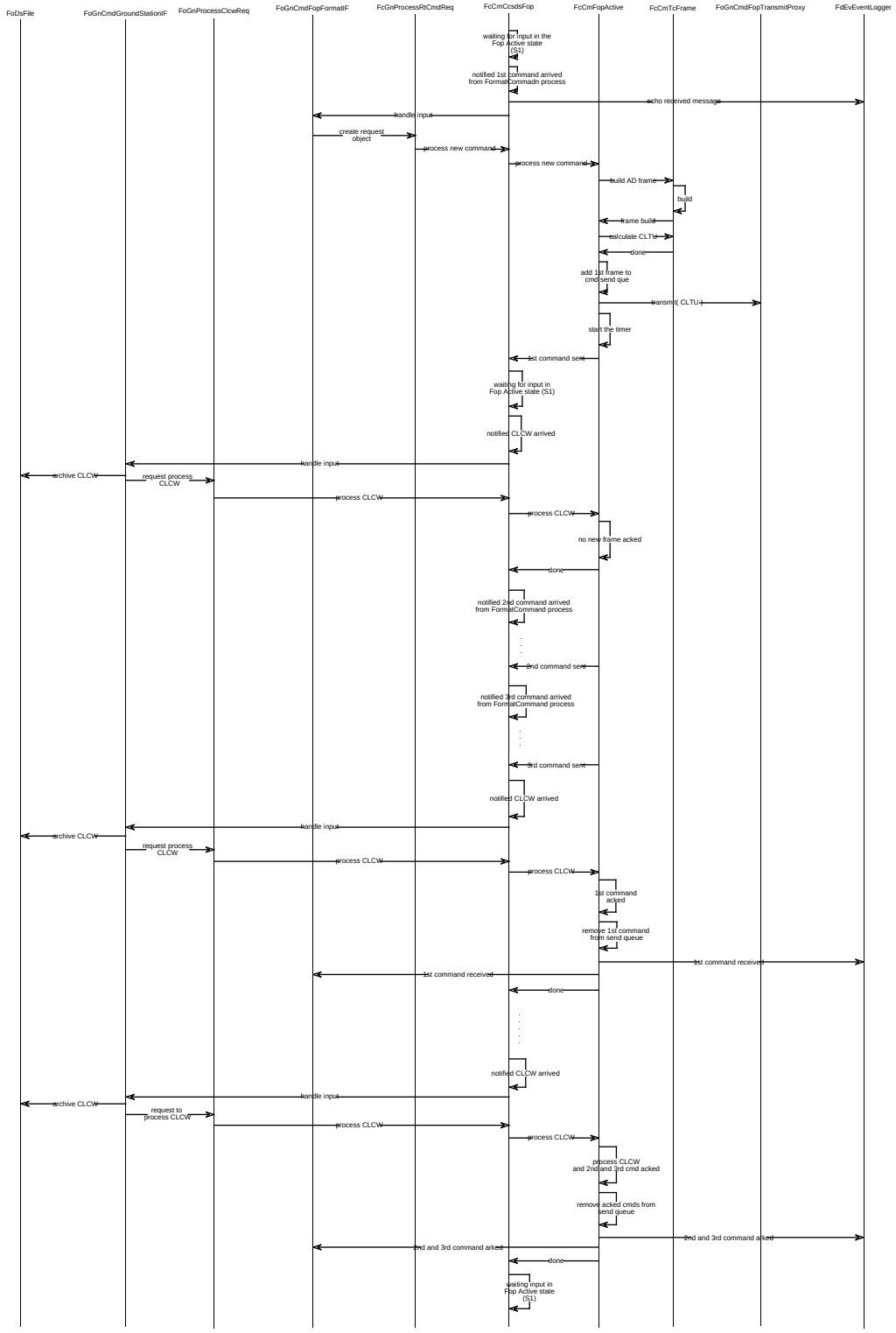

Figure 3.3.4.9-1. FopCommand Transmission scenario

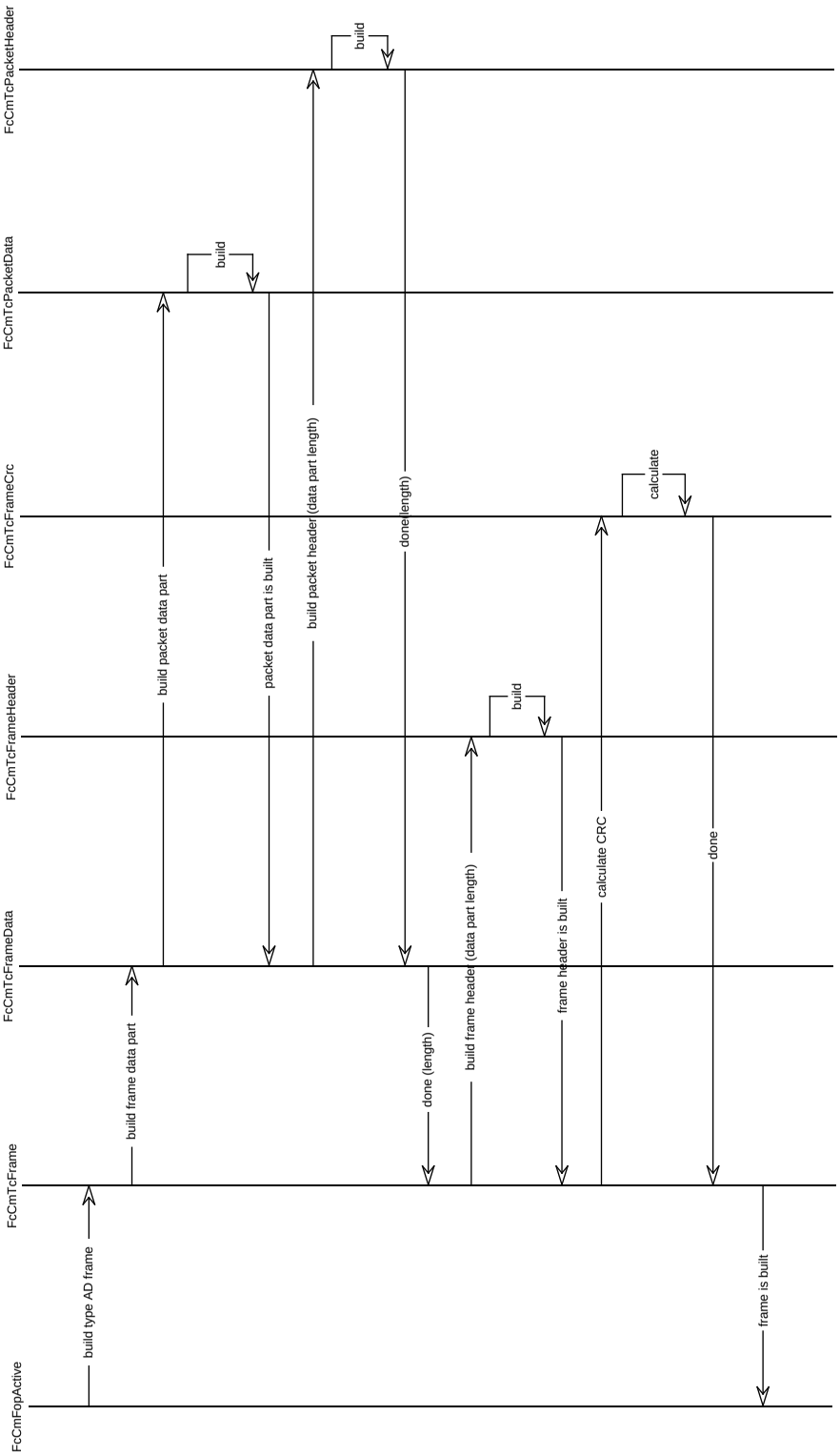

Figure 3.3.4.9-2. FopCommand: Building Transfer Frame

3.3.4.10 Real-Time Command FOP Command Retransmission Scenario

3.3.4.10.1 Real-Time Command FOP Command Retransmission Abstract

The purpose of the "Real-Time Command FOP Command Retransmission " scenario is to describe the process by which the FOP (Frame Operation Procedure) software of the FopCommand process performs a retransmission.

Figure 3.3.4.10-1 is the event trace diagram which correspond to this scenario.

3.3.4.10.2 Real-Time Command FOP Command Retransmission Summary Information

Interfaces:

Data Management Subsystem

EDOS

FormatCommand

TransmitCommand

Stimulus:

A CLCW is received from EDOS.

Desired Response:

CLTUs are retransmitted to the spacecraft.

Pre-Conditions:

FOP is in the "active" state, with three (3) commands in the queue of unconfirmed commands.

Post-Conditions:

The FOP is in the "active" state, with no commands in the queue of unconfirmed commands.

3.3.4.10.3 Scenario Description

The scenario begins with FopCommand waiting for input in the Active state. A CLCW arrives, and FoGnCmdGroundStationIF handles it. The CLCW is archived via FoDsFile, and FcCmProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that the first command sent in the queue was received onboard the spacecraft. The CLTU is removed from the queue of commands sent, and an uplink verification event message is issued via FdEvEventLogger. The FormatCommand process is informed of the successful uplink status of the command via FoGnCmdFopFormatIF, and control is returned to FcCmCcsdsFop in the Active state.

A second CLCW arrives, and FoGnCmdGroundStationIF handles it. The CLCW is archived via FoDsFile, and FcCmProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that retransmission of unconfirmed commands (CLTUs) is required. The two remaining CLTUs in the queue are retransmitted via FoGnCmdFopTransmitProxy and the timer (used for detection of certain retransmission circumstances) is then set, and control is returned to FcCmCcsdsFop in the "Retransmit without wait" state.

A third CLCW arrives, and FoGnCmdGroundStationIF handles it. As before, the CLCW is

archived via FoDsFile, and FoGnProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that the first retransmitted command has been received by the spacecraft. The CLTU is removed from the queue of commands sent, and an uplink verification event message is issued via FdEvEventLogger. The FormatCommand process is informed of the successful uplink status of the command via FoGnCmdFopFormatIF. Inasmuch as it has been confirmed that the retransmission of the queue is at least partially successful, control is now returned to FcCmCcsdsFop, in the Active state.

A fourth CLCW arrives, and FoGnCmdGroundStationIF handles it. Again, the CLCW is archived via FoDsFile, and FoGnProcessClcwReq routes it to FcCmFopActive for processing. The CLCW indicates that the second retransmitted command has been received by the spacecraft. The CLTU is removed from the queue of commands sent, and an uplink verification event message is issued via FdEvEventLogger. The FormatCommand process is informed of the successful uplink status of the command via FoGnCmdFopFormatIF, and control is returned to FcCmCcsdsFop in the Active state.

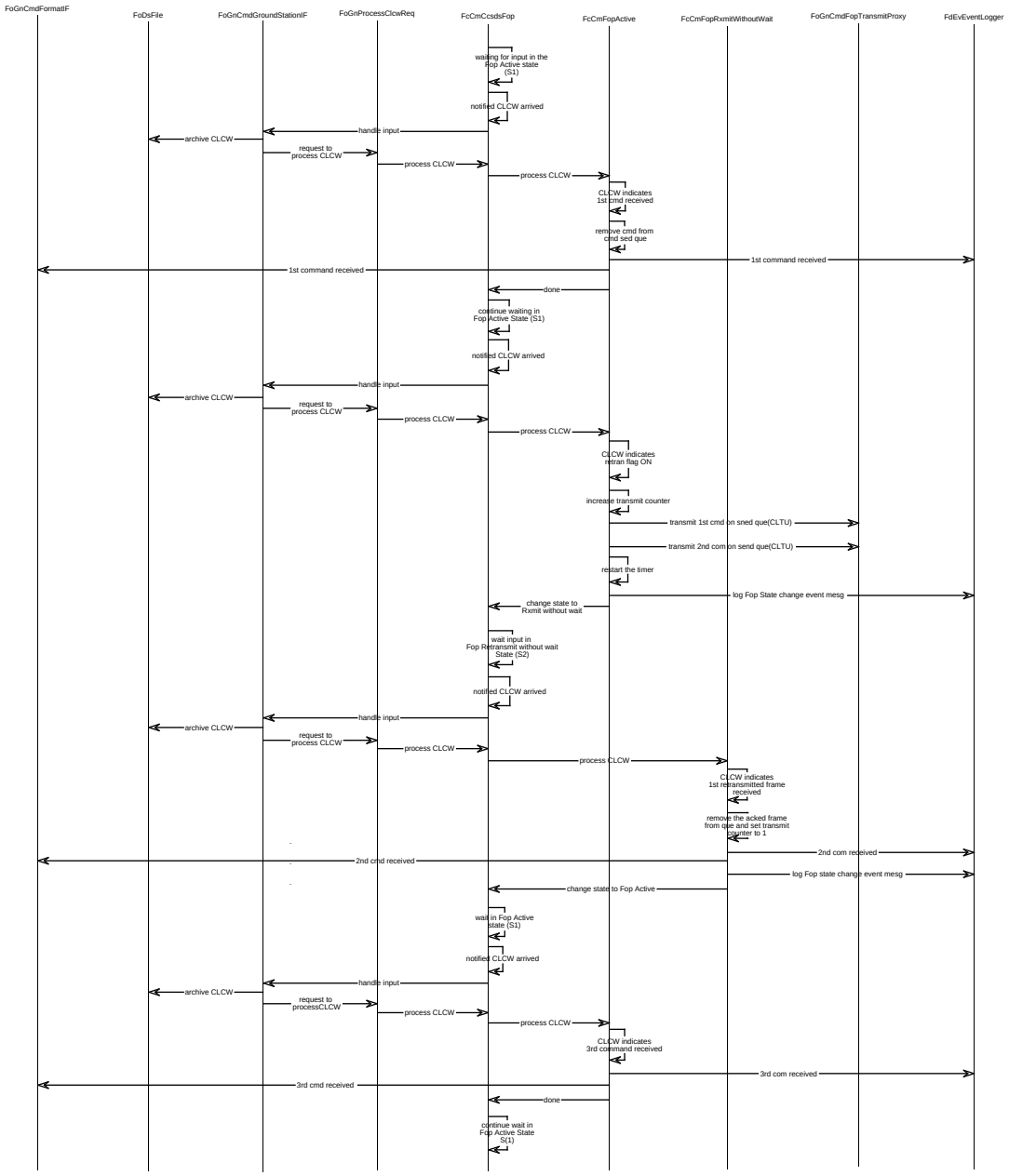

Figure 3.3.4.10-1. FopCommand Retransmission scenario

3.3.4.11FcCmCcsdsFop State Diagram Description:

The dynamic behavior of FcCmCcsdsFop object is best described by a state machine model. At a given time FcCmCcsdsFop object is in one of its six well defined state. When an input arrives or internal event happens, the object analyzes the consequence of these events, takes proper actions and transitions to next state according to a set of per defined rules.

Once initialized, the controller can accept inputs from the following sources: "Transfer command data" request from CommandFormat process (command data may be in the form of 1553B or memory load packet), CLCWs from EDOS, configuration change directives from RMS and internal events which include all the possible exceptions, timeout, and change state request from FcCmFopState.

After initialization the controller is in "Initial" state (S6). It waits for input from outside. In this state, the controller can accept initialization and configuration related directives from RMS or control command (set VR and unlock) from FormatCommand process. All other inputs are either rejected and logged as error or ignored when they are not harmful to the system. When a StartAdWithoutClcw directive received from RMS, the controller does necessary steps to initialize the Fop protocol and transitions to "Active" state. Normal commanding procedure starts. When a StartAdWithClcwCheck directive received from RMS, the controller does necessary steps to initialize the protocol and sets a timer before it transitions to "Initializing without BC Frame" state where it waits for a valid CLCW to show up. When a valid CLCW has been received, Fop protocol transitions to Fop Active state and normal commanding procedure starts. When a SetVr (Set Receiver Frame Sequence number) command is received from FormatCommand process, the controller builds a type BC frame which contains the new sequence number and sends the corresponding CLTU to TransmitCommand process where it is uplinked. The controller then transitions to "Initializing with a BC Frame" state where it waits for the confirmation (by a CLCW) of receipt of the current BC frame. Upon arrival of a CLCW which indicates the BC frame has successfully onboard, the Fop protocol transitions to Fop Active state where normal commanding procedure starts. When an exception occurs during the initialization process, Fop protocol performs a shutdown and transitions to Fop initial state. When an Unlock command received from FormatCommand process, the controller builds a type BC frame and sends the corresponding CLTU to TransmitCommand process where it is uplinked. The controller then transitions to "Initializing with a BC Frame" state where it waits for the confirmation (by a CLCW) of receipt of the BC frame. Upon arrival of a CLCW which indicates the BC frame has successfully onboard, the Fop protocol transitions to Fop Active state where normal commanding procedure starts. When an exception occurs during the process, Fop protocol performs a shutdown and transitions to Fop initial state where it waits for further direction from the operator. When a TerminateAdService directive received from RMS, the controller shutdown the AD service. In Fop initial state, the controller also accepts configuration change requests from RMS. Configuration change requests may include following directives: "Set Fop Sliding Window Size" directive, "Set Ground Transmitter Sequence Number " directive, "Set Transmission Limit" directive and "Set Time Initial Value" directive etc. When those directive are received while Fop is in Initial state, the corresponding actions will be taken and status will be returned to RMS.

The controller is in the "Initializing without BC Frame" state (S4) after receiving a StartAdWithClcw directive while in the "Initial" (S6) state. A successful CLCW check will result in a transmission to "Active" (S1) state where normal commanding procedure starts. When

exceptions or timeout occurs, the controller performs proper shutdown function and transitions back to "Initial" state where it waits for further directive from authority.

The controller is in the "Initializing with BC Frame" state (S5) after receiving an Unlock command or SetVr command while in the "Initial" state. A successful CLCW check will result in a transmission to "Active" state (S1) where normal commanding procedure starts. When timeout occurs while waiting for CLCWs and allowed retry is not exhausted, a retransmission is initiated. When timeout occurs and allowed retry is exhausted, the controller performs a shutdown and returns to "Initial" state (S6) where it waits for further directive form the authority.

Active state (S1) is the normal state of the protocol machine when there are no recent errors on the link and no incidents have occurred leading to flow control problems. In the state, the controller accepts request to transfer command data and CLCWs. Other directives are either flagged as errors when they are not appropriate for the current state or simply ignored when they are not harmful to the system. When a uplink command data request received from FormatCommand process, the controller builds a type AD frame or BC frame depending on the passed command data type according to the CCSDS standard. Once the frame is built, the controller builds the CLTU which is sent to TransmitCommand process where it is uplinked. Before processing any new inputs, the controller saves the current copy of AD or BC frame on the command sent queue. When a CLCW arrives and indicates the current frame has received by the spacecraft, the controller deletes the acknowledged frame from the command sent queue and send status to FormatCommand process. In Fop Active state S(1), the controller also processes CLCWs. When a CLCW indicates several frames have arrived on the spacecraft, the controller removes acknowledged commands from the command sent queue. When CLCW arrives and with the retransmission flag on but wait flag off, the controller initiates retransmission immediately and transitions to "Fop Retransmit Without Wait State" S(2). When CLCW arrives and with both retransmission flag and wait flag on, the controller transitions to "Fop Retransmit With Wait State" S(3). When time out happens while Fop is in its Active state, if the allowed retry is not exhausted, the controller initiates the retransmission, if the allowed retry is exhausted, the controller performs a shutdown and transitions to Fop Initial State and waits for further direction from the operator.

The controller will be in "Retransmit Without Wait" state (S2) if it receives a CLCW with its retransmission flag on but wait flag off while in Fop Active state. When a CLCW arrives and acknowledges some frames, but retransmission is still on and wait flag is still off, the controller removes the acknowledged frames from the command sent queue and initiates another retransmission. If a CLCW arrives with both retransmission flag and wait flag turned off, the controller transitions to Fop Active state S(1) after removal of the acknowledged frame. If a CLCW arrives with its retransmission flag still on and also wait flag on, the controller transitions to "Fop Retransmit With Wait" state S(3). When time out happens while Fop is in the "Retransmit Without Wait" state, if allowed retry is not exhausted, another retransmission is initiated, if allowed retry is exhausted, the controller performs a shutdown and transitions to Fop Initial state where it waits for further direction from the operator.

The controller will be in "Retransmit With Wait" S(3), if it receives a CLCW with its retransmission flag on and also wait flag on while in Fop Active state. When a CLCW arrives with both retransmission and wait flag turned off, the controller transitions to Fop Active state after removal of acknowledged frames from the command sent queue. If a CLCW arrives with its wait flag still on, the controller will stay in this state. When a CLCW arrives with its retransmission flag still on but wait flag off, the controller transitions to "Retransmit Without Wait " state, retransmission starts there.

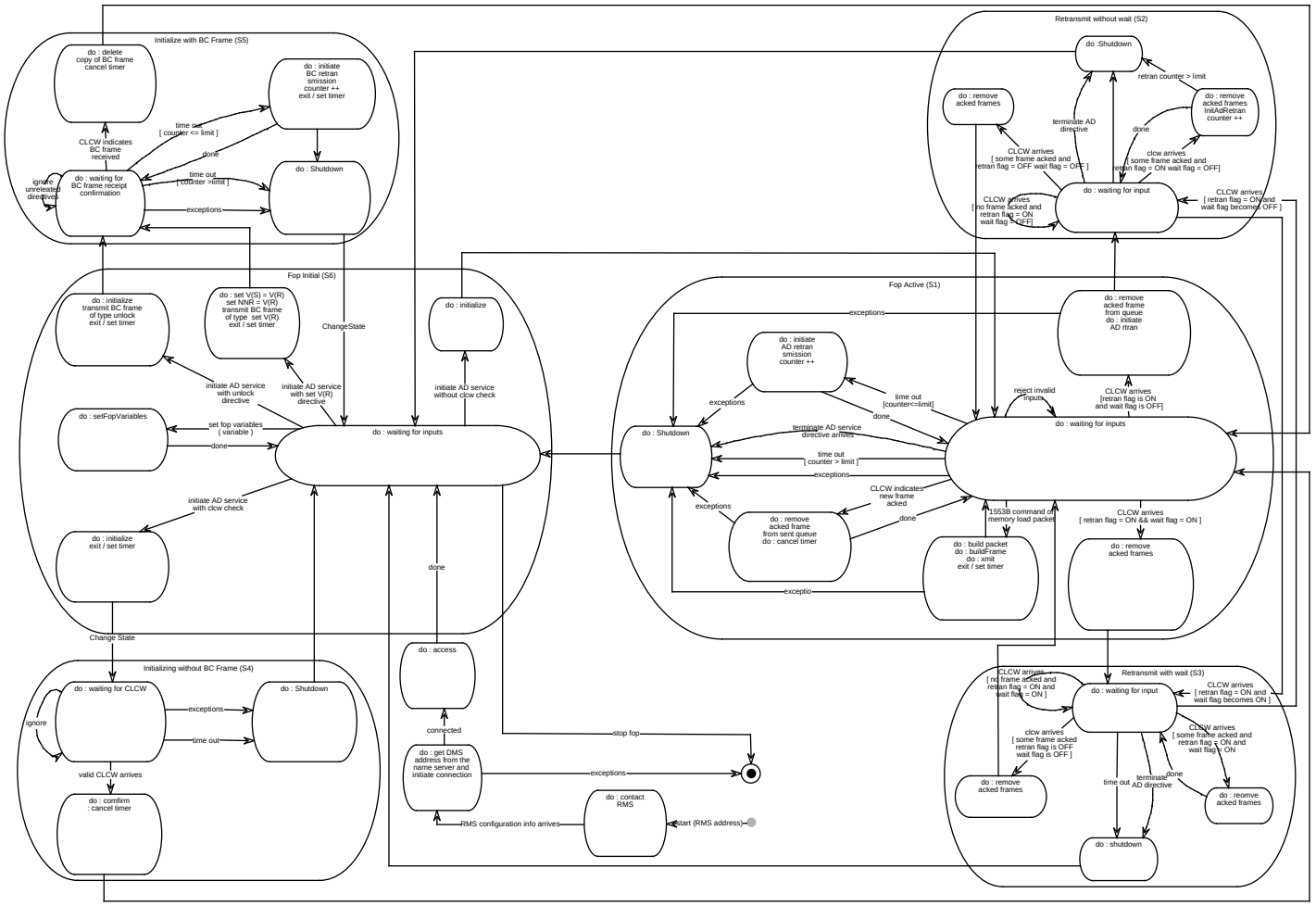

Figure 3.3.4.11-1. FcCmCcsdsFop state diagram

3.3.5 FopCommand Data Dictionary

FcCmCcsdsFop

class **FcCmCcsdsFop**

This class is the controller of the FopCommand process. It establishes connections with other subsystems and command tasks when initialized. It then starts the main event loop. When input arrives from outside, FcCmCcsdsFop interprets the message and delegates the specific request to FcCmFopState. The request is processed by the current active state.

Public Functions

`EcTVoid` **`ChangeRole`**`(RoleType myRole)`

This member function sets attribute myRole to the passed type (Primary, Backup or Inactive). it also calls FcCmFopState::ChangeRole to set the corresponding flag there.

`EcTVoid` **`ChangeState`**`(FcCmFopState*)`

When asked by FcCmFopState class, this member function sets myCurState pointer to next state.

`EcTVoid` **`Configuration`**`(RWCString myConfigInfo)`

This member function does all the necessary configuration stuff using information from the configuration file.

`EcTVoid` **`GetConfigSnapshot`**`()`

This member function delegates the get configuration snapshot request to my current state.

`EcTInt` **`Init`**`(EcTInt argc, EcTChar** argv)`

This routine initializes all the interfaces. It first initiates connection with RMS with passed arguments. Once connected, it waits for configuration information from RMS. When configuration file name is received, it accesses the configuration file via FoDsFile class. It then proceeds to config the controller with info from the file.

`EcTVoid` **`ProcessClcw`**`(EcTUShortInt, EcTBoolean, EcTBoolean, EcTBoolean)`

This member function delegates process CLCW request to my FcCmFopState class. How this request is processed depends on current active state of the FcCmFopState class.

`EcTVoid` **`ProcessLoadPacket`**`(RWCollectable myPacket)`

This member function delegates process a memory load packet request to my current active state. How this request is processed depends on which state is currently active.

`EcTVoid` **`ProcessRtCmd`**`(RWCollectable myRtCmd)`

This member function delegates process a real time

```
command  request
```

to my current active state. How this request is processed depends on which state is currently active.

`EcTVoid` **`ResumeAdService`**`()`

This member function delegates the "Resume AD Service" request to FcCmFopState class. How this request is processed depends on current active state.

`EcTVoid` **`Run`**`()`

This routine starts the main event loop of Fop process. It block waits on all system interfaces. When an input arrives at any of these interfaces, the main event loop will be notified. It will direct the interface to handle the input message. The interface, depending on the context of input, returns an instance of an FoFopRequest object. The FoFopRequest object knows how to ask the controller to perform a specific action.

`EcTVoid` **`SelectCtiu`**`(EcTUInt)`

This member function delegate the select ctiu request to my current state

`EcTVoid` **`SetRetransmissionLimit`**`(EcTUInt myLimit)`

This member function delegates "Set Fop Retransmission Limit" request to my current active state.

`EcTVoid` **`SetTimeoutType`**`(EcTBoolean myTimeoutType)`

This member function delegates "Set Timeout Type" request to my current active state.

`EcTVoid` **`SetTimerInitialVal`**`(EcTULongInt myT1Val)`

This member function delegates "set timer initial value" request to my current active state.

`EcTVoid` **`SetVs`**`(EcTUInt myNewVs)`

This member function delegates the "set Transmitter Frame Sequence Number" request to FcCmFopState. The request is processed by current active state.

`EcTVoid` **`SetWinWidth`**`(EcTUInt myWinWidth)`

This member function delegates the "Set Fop Sliding Window Width" request to my current active state.

`EcTVoid` **`ShutdownFop`**`()`

This member function sets myFopInEffect flag to FALSE;

`EcTVoid` **`StartAdWithClcwCheck`**`()`

This member function delegates RMS "Init AD Service with a CLCW check" request to my FcCmFopState class. How this request will be processed depends on my current active state.

`EcTVoid` **`StartAdWithoutClcwCheck`**`()`

This member function delegates "init AD service without CLCW check" request to my FcCmFopState class. How this request will be processed depends on which state is currently active.

`EcTVoid` **`StartFop`**`()`

This member function sets myFopInEffect flag to TRUE.

`EcTVoid` **`TerminateAdService`**`()`

This routine delegates the terminate AD service request to FcCmFopState. The request will be processed by the current active state.

Private Data

`enum` **`myArchiveState`**

`RWCString` **`myConfigFile`**

This attribute identifies my configuration file name.

`FcCmFopState*` **`myCurState`**

This attributes identifies current Fop active state.

`EcTUInt` **`myDbId`**

This attribute identifies my data base ID.

`FdEvEventLogger*` **`myEventLog`**

This is my pointer to my event log handler FdEvEventLogger class.

`EcTBoolean` **`myFopInEffect`**

This attribute identifies if Fop protocol is running.

`FoGnCmdFopFormatIF*` **`myIfToCmdFormat`**

This is my pointer to FoGnCmdFopFormatIF class.

`FoGnCmdFopGroundStationIF*` **`myIfToGroundStation`**

This is my pointer to FoGnCmdFopGroundStationIF class.

`FoGnCmdFopRmsIF*` **`myIfToRms`**

This is my pointer to FoGnCmdFopRmsIF class.

enum **myRole**

EcTUInt **myScId**

This attribute identifies the spacecraft ID.

Private Types

enum

This attribute identifies my archive state.

Enumerators

OFF
ON

enum

This attribute identifies current string is real-time or a simulation.

Enumerators

RealTime
Simulation

enum

This attribute identifies current string is primary, backup or Inactive.

Enumerators

Backup
Inactive
Primary

FcCmFopActive

class **FcCmFopActive**

This class is a subclass of FcCmFopState. It implements the Fop active state specific behavior.

Base Classes

public **FcCmFopState**

Public Functions

EcTVoid **HandleTimeout**()

When timeout happens while I am in this state, the member function will be called.

EcTVoid **ProcessClcw**(EcTUShortInt, EcTBoolean, EcTBoolean, EcTBoolean)

This member function processes the incoming CLCWs.

EcTVoid **ProcessLoadPacket**(RWCollectable myPacket)

This member function defines the behavior of how Fop protocol processes a "transmit a memory load packet" request while Fop is in its active state.

EcTVoid **ProcessRtcmd**(RWCollectable myRtCmd)

This member function defines the behavior of how Fop protocol processes a "transmit a real time command" request while the protocol is in its active state.

FcCmFopInitial

class **FcCmFopInitial**

This is a subclass of FcCmFopState class. It implements the Fop Initial state specific behavior.

Base Classes

public **FcCmFopState**

Public Functions

EcTVoid **HandleTimeout**()

When timeout happens while Fop is in Initial (S6) state, this function will be called.

EcTVoid **ProcessClcw**(EcTUShortInt EcTBoolean, EcTBoolean, EcTBoolean)

This member function processes CLCWs while Fop in initial state.

EcTVoid **ProcessLoadPacket**(RWCollectable myPacket)

This member function defines the behavior of how the Fop protocol processes a "transmit a memory load packet" request while Fop is in its initial state.

EcTVoid **ProcessRtCmd**(RWCollectable myRtCmd)

This member function defines the behavior of how Fop protocol processes a "transmit a real time command" request while Fop is in initial state.

EcTVoid **ResumeAdService**()

This member function processes "Resume AD service" directive from RMS. It sets Fop current state to previously suspended state.

EcTVoid **SetVs**(EcTUInt myNewVs)

This member function processes "Set transmitter sequence number" request from RMS. It sets myVs attribute to a passed value.

EcTVoid **StartAdWithClcwCheck**()

This member function processes "Initial AD service with a CLCW check" request while Fop is in Initial State. It starts a timer and wait for a valid CLCW to show up.

EcTVoid **StartAdWithoutClcwCheck**()

This member function processes "Init AD service without clcw check" directive from RMS. It sets Fop protocol to its active state and command procedure begins immediately.

FcCmFopInitializeWithBc

class **FcCmFopInitializeWithBc**

This class is a subclass of FcCmFopState. It implements the "Fop Initializing With BC frame" state specific behavior.

Base Classes

public **FcCmFopState**

Public Functions

EcTVoid **HandleTimeout**()

When timeout happens while I am in this state, this member function will be called.

EcTVoid **ProcessClcw**(EcTUShort EcTBoolean, EcTBoolean, EcTBoolean)

This member function defines the behavior of how Fop protocol processes a CLCW while it is in its "Fop Initializing With BC frame" state.

EcTVoid **ProcessLoadPacket**(RWCollectable myPacket)

This member function defines the behavior of how Fop protocol processes a "transmit a memory load packet" request while it is in its "Fop Initializing With BC Frame" state.

EcTVoid **ProcessRtCmd**(RWCollectable myRtCmd)

This member function defines the behavior of how Fop protocol processes a "transmit a real time command" request while the fop protocol is in the "Fop Initializing With BC Frame" state.

FcCmFopInitializeWithoutBc

class **FcCmFopInitializeWithoutBc**

This class is a subclass of FcCmFopState. It implements the "Fop Initializing Without BC frame" state specific behavior.

Base Classes

public **FcCmFopState**

Public Functions

EcTVoid **HandleTimeout**()

When timeout happens while I am in this state, this member function will be called.

EcTVoid **ProcessClcw**(EcTUShort EcTBoolean, EcTBoolean, EcTBoolean)

This member function processes a CLCW while Fop protocol is in "Fop Initializing Without BC frame" state.

EcTVoid **ProcessLoadPacket**(RWCollectable myPacket)

This member function defines the behavior of how Fop protocol processes a "transmit a memory load packet" request while it is in its "Initializing Without BC Frame" state.

EcTVoid **ProcessRtCmd**(RWCollectable myRtCmd)

This member function defines the behavior of how Fop protocol processes "transmit a real time command" request while it is in its "Initializing Without BC frame" state.

FcCmFopRxmitWithWait

class **FcCmFopRxmitWithWait**

This class is a subclass of FcCmFopState. It implements "Fop Retransmit with Wait" state specific behavior.

Base Classes

public **FcCmFopState**

Public Functions

EcTVoid **HandleTimeout**()

When timeout happens while fop protocol is in the "Fop Retransmit with Wait" state, this member function will be called.

EcTVoid **ProcessClcw**(EcTUShortInt, EcTBoolean, EcTBoolean, EcTBoolean)

This member function processes incoming CLCWs while the fop protocol is in the "Fop Retransmit with Wait" state.

EcTVoid **ProcessLoadPacket**(RWCollectable myPacket)

This member function defines the behavior of how the Fop protocol processes a "transmit a memory load packet" request while Fop is in its "Retransmit with Wait" state. This function builds the load packet into a CCSDS frame format and put the frame on the wait queue.

EcTVoid **ProcessRtCmd**(RWCollectable myRtcmd)

This member function defines the behavior of how the Fop protocol processes a "transmit a real time command" request while the protocol is in its Retransmission with Wait state. Because Fop is in Retransmit with Wait state, this function builds the new command into CCSDS frame format and puts it on the wait queue.

FcCmFopRxmitWithoutWait

class **FcCmFopRxmitWithoutWait**

This class is a subclass of FcCmFopState. It implements the "Fop Retransmit without Wait" state specific behavior.

Base Classes

`public` **`FcCmFopState`**

Public Functions

`EcTVoid` **`HandleTimeout`**`()`

When timeout happens while the protocol is in the "Fop Retransmit without Wait", this member function will be called.

`EcTVoid` **`ProcessClcw`**`(EcTUShortInt EcTBoolean, EcTBoolean, EcTBoolean)`

This member function processes CLCWs while the Fop protocol is in the "Fop Retransmit Without Wait" state.

`EcTVoid` **`ProcessLoadPacket`**`(RWCollectable myPacket)`

This member function defines the behavior of how the Fop protocol processes a "transmit a memory load packet" request while the protocol is in its "Retransmission without Wait" state.

`EcTVoid` **`ProcessRtCmd`**`(RWCollectable myRtCmd)`

This member function defines the behavior of how the Fop protocol processes a "transmit real time command" request while the Fop is in its "Retransmission without Wait" state.

FcCmFopState

class **FcCmFopState**

FcCmFopState represents the state of Fop protocol. It has six derived classes with each of them representing a different operational Fop state. FcCmFopState class defines common interface for its six subclasses. It also defines common behavior of the subsequently derived states. When FcCmFopState receives a request from FcCmCcsdsFop, it responds differently depending on its current state.

Public Functions

`EcTVoid` **`AddFrameToSendQue`**`()`

This member function adds frame to myCmdSendQue before sends the corresponding cltu to command transmit process.

`FcGnTcCltu` **`BuildFrame`**`(RWCollectable myCmdData)`

This member function asks FcCmTcFrame class to build a 1553B command or a memory load packet into Type-AD or Type-BC frame according to the CCSDS standard. It calls FcCmTcFrame::BuildFrame to actually build the frame. FcCmTcFrame::BuildFrame is overloaded, therefor, with the passed argument properly casted, it knows to handle a command in 1553B format or a memory load packet differently. This member function returns an instance of FcGnTcCltu.

`EcTVoid` **`ChangeRole`**`(RoleType myRole)`

This routine sets myRole to the passed value.

`EcTInt` **`Config`**`(RWSet myParameterSet)`

This member function gets the passed configuration parameter set and does configuration.

`RWCString` **`GetConfigSnapshot`**`()`

This routine saves my configuration parameters in a configuration file and notify RMS about the file.

`virtual EcTVoid` **`HandleTimeout`**`()`

This is a virtual function. It provides common interface for all subsequently derived classes.

`EcTVoid` **`Ignore`**`()`

Sometimes, it is not necessary to process the request that I've received. This method handles this kind of situation.

```
EcTInt Init(FcCmCcsdsFop* myFop, FoCdFopFormatProxy* myFormatProxy,
      FoGnCmdFopRmsIF* myPtrToRms, FdEvEventLogger* myEventLog)
```

Initialize the fop states and interface pointers.

`EcTVoid` **`InitiateAdRetran`**`()`

this member function prepares for AD or BC frame retransmission.

`EcTVoid` **`InitiateFrameTransmit`**`()`

InitiateFrameTransmit()

This member function initiates the transmission of the first frame on the command send queue that has its To Be Retransmitted flag on. If no frames are needed to be retransmitted, this member function initiates the transmission of a new frame.

`virtual EcTVoid` **`ProcessClcw`**`(EcTUShortInt, EcTBoolean, EcTBoolean, EcTBoolean)`

This is a virtual function. It provides common interface for all the subsequently derived classes.

`virtual EcTVoid` **`ProcessLoadPacket`**`(RWCollectable myPacket)`

This is a virtual function. It provides default behavior for how FcCmFopState processes a memory load packet. If subsequent derived class wants to process a load packet differently, it should provides its own version of ProcessLoadPacket function.

`virtual EcTVoid` **`ProcessRtCmd`**`(RWCollectable myRtCmd)`

This is a virtual function. It provides default behavior for how FcCmFopState processes a real time command. If subsequent derived class wants to process a command differently, it should provide its own version of ProcessRtCmd function.

`EcTVoid` **`RemoveAckedFrame`**`()`

This member function removes acked frames from command send queue.

`virtual EcTVoid` **`ResumeAdService`**`()`

This is a virtual function. It provides the default behavior for how the FcCmFopState handles the "Resume AD service" request. The derived class FcCmFopInitial will override this function.

`EcTVoid` **`SelectCtiu`**`(EcTUInt myCtiu)`

This member function sets my attribute myCtiu.

`EcTInt` **`SetTimeHandler`**`()`

This member function sets a time out handler.

`EcTVoid` **`SetTimeoutType`**`(EcTBoolean myTimeoutType)`

This member function sets Fop's Timeout Type variable to a passed value.

`EcTVoid` **`SetTimerInitialVal`**`(EcTULongInt myT1Val)`

This member function sets Fop's T1 value to a passed number

`EcTVoid` **`SetTransmissionLimit`**`(EcTUInt myLimit)`

This member function sets Fop's Transmission limit variable to a passed number.

`virtual EcTVoid` **`SetVs`**`(EcTUInt myNewVs)`

This is a virtual function. It provides the default behavior for how the FcCmFopState handles the "set transmitter frame sequence number" request. The derived class FcCmFopInitial will override this function

`EcTVoid` **`SetWinWidth`**`(EcTUInt myWinWidth)`

This member function sets Fop Sliding Window Width to a passed value.

`EcTVoid` **`Shutdown`**`()`

This routine is responsible for gracefully shutting down FOP whenever needed.

`virtual EcTVoid` **`StartAdWithClcwCheck`**`()`

This is a virtual function. It provides default behavior for how FcCmFopState processes a "Start AD Service With a CLCW check" request. The derived class FcCmFopInitial will override this function.

`virtual EcTVoid` **`StartAdWithoutClcwCheck`**`()`

This is a virtual function. It provides the default behavior for how the FcCmFopState handles the StartAdWithoutClcwCheck request. The derived class FcCmFopInitial should override this function.

`EcTVoid` **`StartTimer`**`()`

This member function starts system timer.

`EcTVoid` **`StopTimer`**`()`

This member function stops system timer.

`EcTVoid` **`TerminateAdService`**`()`

This routine terminates AD service gracefully and informs all the related parties about the termination.

`EcTVoid` **`TransmitFrame`**`(FcGnTcCltu myCltu)`

This member function does preparation for transmission, and sends a copy of cltu to command transmit task via its proxy.

Private Data

`RWlistCollectablesQueue` **`myCmdSentQue`**

This attribute contains all the transfer frames that have been uplinked but not CLCW verified.

`RWlistCollectablesQueue` **`myCmdWaitQue`**

This attribute contains one command that will be processed next.

`enum` **`myCtiu`**

`FcCmTcFrame` **`myCurFrame`**

This attribute identifies the current copy of type AD or BC frame.

`FdEvEventLogger*` **`myEventLog`**

This attribute is my pointer to Event log class FdEvEventLogger.

`EcTUShortInt` **`myExpectedAckSeqNo`**

This attribute identifies the Expected Acknowledgment Frame Sequence Number, NN(R). The NN(R) contains the value of N(R) from the previous CLCW. NN(R) -1 is the value of the sequence number of the latest Type-AD frame which Fop can guarantee has arrived safely.

`FcCmCcsdsFop*` **`myFop`**

This is my pointer to FcCmFopCcsdsFop class.

`FcCdFopForamtProxy*` **`myFormatProxy`**

This attribute identifies my pointer to Command Format task.

`EcTUShortInt` **`myFrameSeqNo`**

This attribute identifies my frame sequence number.

`FoPsClientIF*` **`myParaServerProxy`**

This is my pointer to parameter server.

`FoGnCmdFopRmsIF*` **`myPtrToRms`**

This attribute is my pointer to RMS subsystem.

`enum` **`myRole`**

`enum` **`myStoredLockoutFlag`**

`enum` **`myStoredRetranFlag`**

`enum` **`myStoredWaitFlag`**

`EcTUInt` **`mySuspendState`**

This attribute identifies Fop suspend state.

`EcTBoolean` **`myTimeoutType`**

This attribute the Fop Timeout Action, i.e. when timeout happens, what action Fop will take.

`EcTULongInt` **`myTimerInitialVal`**

this attribute identifies the timeout period.

`enum` **`myToBeRetranFlag`**

`EcTUInt` **`myTransmitCounter`**

This attribute identifies how many time a frame has been transmitted.

`EcTUInt` **`myTransmitLimit`**

This attribute identifies how many time Fop can retry.

`FcCmCmdFopTransmitProxy*` **`myTransmitProxy`**

This attribute identifies my pointer to Command Transmit task.

`EcTUInt` **`myVs`**

This attribute identifies grounder transmitter sequence number.

`EcTUInt` **`myWinWidth`**

This attribute identifies Fop Sliding Window Width.

Private Types

`enum`

This attribute identifies which CTIU identifier to use when build the Frames.

Enumerators

`Backup`
`Primary`

`enum`

This attribute identifies the current process is a backup, primary or Inactive.

Enumerators

`Backup`
`Inactive`
`Primary`

`enum`

This attribute identifies the a frame on the command send queue that must be retransmitted.

Enumerators

`OFF`
`ON`

`enum`

This attribute contains the value of the "Retransmit" flag from the previous CLCW.

Enumerators

`OFF`
`ON`

`enum`

This attribute contains the value of the "Wait" flag from the previous CLCW.

Enumerators

OFF
ON

enum

This attribute identifies the value of the "Lockout" flag from the previous CLCW.

Enumerators

OFF
ON

FcCmTcFrame

class **FcCmTcFrame**

This class is responsible for building transfer frame according to the CCSDS format.

Public Functions

FcCmTcCltu **BuildCltu**()

This member function calculates the cltu for the entire transfer frame. It creates an instance of FcCmTcCltu class.

EcTInt **BuildFrame**(FcGnFopPacketMsg myPacket)

This member function builds a memory load packet (in CCSDS packet format) into CCSDS frame format.

EcTInt **BuildFrame**(FcGnFopCmdMsg myRtCmd)

This member function builds a real time command (in 1553B format) into the CCSDS frame format;

EcTVoid **SetUplinkStatus**(EcTBoolean)

This member function sets myUplinkstatus attribute.

Private Data

EcTUChar* **myCltu**

This attribute identifies my cltu.

FcCmTcFrameCrc* **myCrcPtr**

This is my pointer to FcCmFrameCrc class.

EcTUChar* **myFrame**

This is the pointer to my transfer frame.

EcTUShortInt **myFrameSeqNo**

This attribute identifies the current frame sequence number.

enum **myFrameType**

FcCmTcFrameHeader* **myHeaderPtr**

This is my pointer to FcCmTcFrameHeader class.

FcCmTcFramePacket* **myPacketPtr**

This is my pointer to FcCmTcFramePacket class.

enum **myToBeRetransmittedFlag**

enum **myUplinkStatus**

Private Types

enum

This attribute identifies my transfer frame type.

Enumerators

AD
BC

enum

This attribute identifies up link status of my frame.

Enumerators

bad
good

enum

This attribute identifies if my frame need to be retransmitted.

Enumerators

off
on

FcCmTcFrameCrc

class **FcCmTcFrameCrc**

This class is responsible for calculating CRC code for frame.

Public Functions

EcTInt **Build**(EcTUChar*)

This member function calculates CRC code for frame.

EcTVoid **BuildCrcTable**()

This member function builds CRC table.

Private Data

EcTUInt **myCrcPoly**

This attribute identifies CRC generating poly.

EcTUInt* **myCrcTable**

This attribute identifies crctable.

EcTUInt **myCrcVal**

This attribute identifies final CRC value

EcTUchar* **myFrame**

This attribute identifies the TC frame.

FcCmTcFrameHeader

class **FcCmTcFrameHeader**

This class is responsible for building frame header.

Public Functions

EcTInt **Build**(EcTUChar*)

This member function builds the header

EcTUInt **GetBits**(EcTUInt, EcTUInt)

This member function gets number of bits from a given starting position.

EcTVoid **SetBits**(EcTUInt, EcTUInt)

This member function sets number of bits from a given starting position.

Private Data

EcTUChar* **myBufferPtr**

This attribute identifies the buffer pointer,

enum **myBypassFlag**

enum **myControlCmdFlag**

EcTUInt **myCtiuIdentifier**

This attribute identifies my ctiu

EcTUInt **myLength**

This attribute identifies my header length.

EcTUInt **myOffSet**

This attribute identifies my offset in my frame.

EcTUInt **myScId**

This attribute identifies my space craft Id.

EcTUShortInt **mySequenceNo**

This attribute identifies the current frame sequence number.

EcTUInt **myVersionNo**

This attribute defines the version number of the TC frame.

EcTUInt **myVirtualChannelId**

This attribute identifies virtual channel.

Private Types

enum

This attribute defines control command flag.

Enumerators

OFF
ON

enum

This attribute defines myBypassFlag which controls the application of "Frame Acceptance Checks".

Enumerators

OFF
ON

FcCmTcFramePacket

class **FcCmTcFramePacket**

This class is responsible for building frame data part.

Public Functions

EcTInt **Build**(EcTUChar*)

This member function builds frame data part.

Private Data

`FcCmTcPacletData* **myDataPtr**`

This is my pointer to FcCmTcPacketData class.

`EcTUInt **myLength**`

This attribute identifies my frame data length.

`EcTUInt **myOffset**`

This attribute identifies my data part offset from frame header.

`FcCmTcPacketHeader* **myPacketHdPtr**`

This is my pointer to FcCmPacketHeader class.

`EcTUChar* **myPacketPtr**`

This attribute identifies my packet.

FcCmTcPacketData

class **FcCmTcPacketData**

This class is responsible for building the packet data part.

Public Functions

`EcTInt **Build**(EcTUChar*)`

This member function builds packet data part,

Private Data

`EcTUChar* **myBufferPtr**`

this attribute identifies the buffer pointer.

`EcTUInt **myLength**`

This attribute identifies the length of my packet.

`EcTUInt **myOffset**`

This attribute identifies the offset from my packet header.

FcCmTcPacketHeader

class **FcCmTcPacketHeader**

This class is responsible for building packet header.

Public Functions

`EcTInt **Build**(EcTUChar*)`

This member function builds packet header.

`EcTUInt **GetBits**(EcTUInt, EcTUInt)`

This member function gets given umber of bits from a given starting point.

`EcTVoid **SetBits**(EcTUInt, EcTUInt)`

This member function sets given number of bits from

```
a given starting point.
```

Private Data

`EcTUInt **myApid**`

This attribute identifies application process identifier.

`EcTUChar* myBufferPtr`

This attribute identifies my buffer pointer.

`EcTUInt myLength`

This attribute identifies my packet length.

`EcTUInt myOffset`

This attribute identifies packet offset in the frame.

`EcTUShortInt myPacketSeqNo`

This attribute identifies packet sequence no.

`EcTUInt myPacketType`

This attribute identifies my packet type.

`EcTUInt mySecondaryHeaderFlag`

This attribute identifies Secondary Header flag

`EcTUInt mySequenceFlag`

This attribute identifies sequence flag

`EcTUInt myVersionNo`

This attribute identifies version member of a packet.

FcGnFormatProcessReq

class FcGnFormatProcessReq

This class provides a common interface for all the requests from FormatCommand subsystem.

Base Classes

`public FoFopRequest`

Public Functions

`virtual EcTInt Execute(FcCmCcsdsFop* fop)`

This member function provides a common interface for all requests and will be overridden by sub class method.

FcGnProcessLoadPacketReq

class FcGnProcessLoadPacketReq

This class defines a binding between "process load packet" request of FormatCommand process and the ProcessLoadPacket operation of the controller.

Base Classes

`public FcGnFormatProcessReq`

Public Functions

`EcTInt Execute(FcCmCcsdsFop* fop)`

This member function invokes the ProcessLoadPacket function of the controller.

Private Data

`RWCollectable myPacket`

This attribute identifies the load packet sent by FormatCommand process.

FcGnProcessRtCmdReq

class **FcGnProcessRtCmdReq**

This class defines a binding between "Process real time command" request of FormatCommand process and the ProcessRtCmd operation of the controller class.

Base Classes

public **FcGnFormatProcessReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes ProcessRtCmd function of the controller class.

Private Data

RWCollectable **myRtCmd**

This attribute identifies the real time command sent by FormatCommand process.

FoCmCCSDSFopProxy

class **FoCmCCSDSFopProxy**

This class is a proxy class that FopCommand process provides for FormatCommand process. FormatCommand process sends real time commands and memory load packets to FopCommand class via this class.

Public Functions

EcTBoolean **ProcessLoadPacket**(EcTUChar*, EcTInt, FcTCdLoadStage, RWCString)

This member function asks FopCommand process to process a memory load packet.

EcTBoolean **ProcessRtCmd**(EcTUChar*, EcTInt, EcTBoolean)

This member function asks FopCommand process to process a real time command.

FoFopRequest

class **FoFopRequest**

The class FoFopRequest is an abstract class, it is not intended to be instantiated. It provides a common interface for all subsequently derived classes.

Public Functions

virtual EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function provides interface and will be overridden by sub class member function.

FoGnChangeRoleReq

class **FoGnChangeRoleReq**

This class defines a binding between "change operational state" request of RMS and the ChangeRole function of the controller class.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes ChangeRole function of controller.

Private Data

`RoleType` **`myRole`**

This attribute identifies if the current string is in Primary, Backup or Inactive state.

FoGnCmdFopFormatIF

`class` **FoGnCmdFopFormatIF**

FoGnCmdFormatIF

This class facilitates the exchange of information between command Fop process and command format process

Public Functions

`RWCollectable*` **`HandleInput`**`()`

This routine reads message from input stream. Depends on the context of the input, It creates an instance of ProcessRtCmd object or an instance of ProcessLoadPacket object, each object knows how to invoke certain actions of the controller.

`EcTInt` **`InitFormatIf`**`(RWCString myFormatAddress)`

This member function initializes Fop's interface to command format process.

Private Data

`FdEvEventLogger*` **`myEventLog`**

This attributes points to my event handle class FdEvEventLogger.

`RWCString` **`myFormatAddress`**

This attribute identifies the command format process address.

`EcTInt` **`myListenPort`**

This attribute identifies my listening port.

FoGnCmdFopGroundStationIF

`class` **FoGnCmdFopGroundStationIF**

This class facilitates the exchange of information between command fop process and the ground station.

Public Functions

`RWCollectable*` **`HandleInput`**`()`

This routine reads message from input stream. It returns an instance of FoGnProcessClcwReq which knows how to invoke process clcw function of the controller.

`EcTInt` **`InitGroundStaionIF`**`(RWCString)`
`EcTInt` **`SetNotifier`**`()`

InitGroundStationIF

This member function initializes Fop's interface to ground station

Private Data

`FdEvEventLogger*` **`myEventLog`**

This attribute points to my event handle class FdEvEventLogger.

`RWCString` **`myGroundStationadd`**

This attribute identifies my ground station address.

FoGnCmdFopRmsIF

class **FoGnCmdFopRmsIF**

this class facilitates the exchanges of information between command fop process and RMS subsystem.

Public Functions

`RWCollectable* **HandleInput**()`

This routine reads message from input stream. Depends on the context of the input, it returns an instance of FoFopRequest which will have intelligence to ask the controller to perform certain actions.

`EcTInt **Init**(EcTInt argc, EcTChar** argv)`

This routine initializes Fop's interface to RMS subsystem. It establishes the connection and registers the connection in an event notifier.

`EcTVoid **SendStatus**(RWCString)`

This member function sends status back to RMS subsystem.

Private Data

`FdEvEventLogger* **myEventLog**`

this attribute gives this class the visibility to FdEvEventLogger Class

`RWCString **myRmsAddress**`

this attribute identifies the address of the RMS subsystem.

FoGnCmdFopRmsProxy

class **FoGnCmdFopRmsProxy**

This class is a proxy that FopCommand provides for RMS subsystem. RMS sends configuration related directives via this class.

Public Functions

`EcTVoid **ChangeRole**(RoleType myRole)`

This member function asks FopCommand process to change its operational state. (the state is identified as role in FopCommand process, the role can be Primary, Backup or Inactive);

`EcTVoid **ConfigFopCommand**(RWCString myConfigMsg)`

This member function sends process configuration information to FopCommand process.

`EcTVoid **GetConfigSnapshot**()`

This member function asks FopCommand process to take a snap shot.

`EcTVoid **ResumeAd**()`

This member function asks FopCommand process to resume its AD service.

`EcTVoid **SelectCtiu**(EcTUInt myCtiu)`

This member function asks FopCommand process to set its CTIU to the passed value.

`EcTVoid **SetTimeInitialVal**(EcTULongInt myT1Val)`

This member function asks the FopCommand process to set its T1 initial value to the passed value.

`EcTVoid **SetTimeoutType**(EcTBoolean myTimeoutType)`

This member function asks the FopCommand process to set its timeout type value which specifies what action to take if timeout happens.

`EcTVoid **SetTransmissionLimit**(EcTUInt myLimit)`

This member function asks the FopCommand process to set its transmission limit to the passed value.

`EcTVoid` **`SetVs`**`(EcTUInt myVs)`

This member function asks FopCommand process to set its ground transmitter frame sequence number to the passed value.

`EcTVoid` **`SetWinWidth`**`(EcTUInt myWinWidth)`

This member function asks the FopCommand process to set its sliding window width to the passed value.

`EcTVoid` **`ShutdownFop`**`()`

This member function asks the FopCommand process to shut down itself.

`EcTVoid` **`StartAdWithClcwCheck`**`()`

This member function asks FopCommand process to start AD service with a CLCW check.

`EcTVoid` **`StartAdWithoutClcw`**`()`

This member function asks FopCommand process to start AD service without a CLCW check.

`EcTVoid` **`TerminateAd`**`()`

This member function asks FopCommand process to terminate its AD service.

FoGnCmdFopTransmitProxy

`class` **`FoGnCmdFopTransmitProxy`**

This class facilitates the exchange information between FopCommand process and TransmitCommand process.

Public Functions

`EcTBoolean` **`SendCltu`**`(FcGnTcCltu myCltu)`

This member function will send cltu object to TransmitCommand process where the command will be uplinked.

FoGnGetConfigSnapshotReq

`class` **`FoGnGetConfigSnapshotReq`**

This class defines a binding between RMS "Get Configuration snapshot" request and the GetConfigSnapshot operation of FcCmCcsdsFop class.

Base Classes

`public` **`FoGnRmsReq`**

Public Functions

`EcTInt` **`Execute`**`(FcCmCcsdsFop* fop)`

This member function invokes the GetConfigSnapshot operation of FcCmCcsdsFop class.

Private Data

`RWCString` **`myFileName`**

This attribute identifies the configuration file name.

FoGnProcessClcwReq

`class` **`FoGnProcessClcwReq`**

This class does preliminary CLCW process. It then requests FcCmCcsdsFop to take certain action based on its current state.

Base Classes

public **FoFopRequest**

Public Functions

EcTVoid **ArchiveClcw**()

This member function archives CLCW to a file on an hourly basis.

EcTInt **DeComClcw**()

This member function decommutate CLCW.

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes ProcessClcw function of FcCmCcsdsFop.

EcTInt **ValidateClcw**()

Validate

This member function validates CLCW bit pattern according to CCSDS standard.

Private Data

EcTUInt **myCurClcw**

This attribute identifies my CLCW.

EcTBoolean **myLockFlag**

This attribute identifies the lock out flag of a CLCW. When this flag is on, all subsequent type A BC frames are rejected.

EcTUShortInt **myNextExpectedSeqNo**

This attribute identifies FARM's next expected frame sequence number.

EcTBoolean **myRetranFlag**

This attribute identifies the retransmission flag of a CLCW. When this flag is on, retransmission is required.

EcTBoolean **myWaitFlag**

This attribute identifies the wait flag of a CLCW. When the flag is on, this indicates the spacecraft is unable to pass data to the higher layer.

FoGnResumeAdServiceReq

class **FoGnResumeAdServiceReq**

This class defines a binding between "Resume AD Service" request and the ResumeAdService operation of my controller.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes the ResumeAdService operation of FcCmCcsdsFop class.

EcTInt **SetSuspendState**(EcTUInt myState)

This member function sets attribute mySuspendState to passed value.

Private Data

EcTUInt **mySuspendState**

This attribute identifies Fop suspend state.

FoGnRmsReq

class **FoGnRmsReq**

This class provides a common interface for all subsequently derived RMS request classes.

Base Classes

`public` **FoFopRequest**

Public Functions

`virtual EcTInt` **Execute**`(FcCmCcsdsFop* fop)`

This member function provides a common interface for all RMS requests and will be overridden by sub class method.

`EcTInt` **SetDirective**`()`

This member function sets myDirective to the incoming one.

Private Data

`RWCString` **myDirective**

This attribute identifies the received directive

FoGnSelectCtiu

class **FoGnSelectCtiu**

This class defines a binding between RMS "select ctiu" request and SelectCtiu operation of FcCmCcsdsFop class.

Base Classes

`public` **FoGnRmsReq**

Public Functions

`EcTInt` **Execute**`(FcCmCcsdsFop* fop)`

This member function invokes SelectCtiu operation of FcCmCcsdsFop class.

Private Data

`EcTUInt` **myCtiu**

This attribute identifies my ctiu

FoGnSetRetransmissionLimitReq

class **FoGnSetRetransmissionLimitReq**

This class defines a binding between "Set Retransmission Limit" request and SetRetransmissionLimit operation of my controller class.

Base Classes

`public` **FoGnRmsReq**

Public Functions

`EcTInt` **Execute**`(FcCmCcsdsFop* fop)`

This member function invokes SetRetransmissionLimit operation of my controller class through passed pointer.

`EcTInt` **SetRetransmissionLimit**`(EcTUInt myLimit)`

This member function sets data member value.

Private Data

`EcTUInt` **`myLimit`**

This attribute identifies transmission limit of Fop.

FoGnSetTimeInitialValReq

`class` **`FoGnSetTimeInitialValReq`**

This class defines a binding between "Set Timer Initial Value" request and SetTimerInitialVal operation of my controller class.

Base Classes

`public` **`FoGnRmsReq`**

Public Functions

`EcTInt` **`Execute`**`(FcCmCcsdsFop* fop)`

This member function invokes SetTimerInitialVal operation of my controller class through passed pointer.

`EcTInt` **`SetTimeIitialVal`**`(EcTULongInt myT1Val)`

SetTimeInitialVal

This member function sets myT1Val attribute to passed value.

Private Data

`EcTULongInt` **`myT1Val`**

This attribute identifies my initial timeout value.

FoGnSetTimeoutTypeReq

`class` **`FoGnSetTimeoutTypeReq`**

This class defines a binding between "Set Timeout Type" request and SetTimeoutType method of FcCmCcsdsFop class.

Base Classes

`public` **`FoGnRmsReq`**

Public Functions

`EcTInt` **`Execute`**`(FcCmCcsdsFop* fop)`

This member function invokes SetTimeoutType operation of my controller class through passed pointer.

`EcTInt` **`SetTimeoutType`**`(EcTBoolean myTimeoutType)`

This member function sets myTimeoutType attribute.

Private Data

`EcTBoolean` **`myTimeoutType`**

This attribute identifies what action fop will take when timeout happens.

FoGnSetVsReq

`class` **`FoGnSetVsReq`**

This class defines a binding between "Set Transmitter Frame Sequence Number" request and SetVs operation of my controller.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes SetVs function of controller class.

EcTInt **SetVs**(EcTUInt myVs)

This member function sets attribute myNewVs to passed value.

Private Data

EcTUShortInt **myNewVs**

This attribute identifies my new transmitter frame sequence number.

FoGnSetWinWidthReq

class **FoGnSetWinWidthReq**

This class defines a binding between "Set Fop Sliding Window Width" request and SetWinWidth operation of my controller.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes SetWinWidth operation of the controller class through passed pointer.

EcTInt **SetWinWidth**(EcTUInt myWinWidth)

This member function sets myWinWidth attribute to the passed value.

Private Data

EcTUInt **myWinWidth**

This attribute identifies Fop Sliding Window Width.

FoGnShutdownFopReq

class **FoGnShutdownFopReq**

This class defines a binding between RMS request "shot down" and FcCmCcsdsFop::ShutdownFop member function.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes FcCmCcsdsFop::ShutdownFop function of the controller.

FoGnStartAdWithClcwCheckReq

class **FoGnStartAdWithClcwCheckReq**

This class defines a binding between RMS directive "Start AD Service with a CLCW check" and StartAdWithClcwCheck method of FcCmCcsdsFop class.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes StartAdWithClcwCheck method of the controller class.

FoGnStartAdWithoutClcwReq

class **FoGnStartAdWithoutClcwReq**

This class defines a binding between RMS directive "Init Ad service without clcw check" request and StartAdWithoutClcwCheck operation of FcCmCcsdsFop class.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes StartAdWithoutClcwCheck operation of FcCmCcsdsFop class.

FoGnTerminateAdReq

class **FoGnTerminateAdReq**

This class defines a binding between RMS "Terminate AD Service" request and TerminateAdService operation of FccmCcsdsFop.

Base Classes

public **FoGnRmsReq**

Public Functions

EcTInt **Execute**(FcCmCcsdsFop* fop)

This member function invokes TerminateAdService operation of FcCmCcsdsFop class.

3.4 TransmitCommand Description

The TransmitCommand process receives Command Link Transfer Units (CLTUs) from the FopCommand process and sends them to EDOS at a data rate corresponding to the current uplink bandwidth (10, 2, 1 or .125 kbps). TransmitCommand uses the uplink path as defined by the channel (SSA, SMA, S-Band) and spacecraft antenna (HighGain, Omni) to determine the data rate.

3.4.1 TransmitCommand Context Description

The context diagram in Figure 3.4.1-1 depicts the data flows between the Transmit Command process, the internal EOC and external ground system components. Descriptions of data flows are summarized for each component:

FOS Resource Management Subsystem (RMS): RMS starts the TransmitCommand process running as part of a logical string and then supplies EOC spacecraft contact and commanding session configuration information. This information includes address of RMS Subsystems, and TransmitCommand database ID, spacecraft ID, state (i.e., primary or backup), operational mode (real-time or simulation), addresses of the Parameter Server and the FopCommand process. Additionally, it is responsible for managing archive mode (on, off) and uplink path configuration directives (which specify the channel and spacecraft antenna being used) from User Interface in a manner that insures the backup TransmitCommand task will be properly configured (i.e., "hot") to take over processing in the event of a failure scenario involving the primary TransmitCommand process.

FOS Data Management Subsystem (DMS) : This subsystem receives, stores and forwards to appropriate subsystems the TransmitCommand event messages, and uplinked commands (CLTUs). Configuration files, both standard-startup and snapshot, are written to and read from the DMS. The configuration file includes the uplink path information (Channel and current spacecraft Antenna specification), and command archive information (Archive State).

Parameter Server: The parameter server is responsible for distributing new parameter values (i.e., ground telemetry) to those processes which have requested (i.e.,registered) to be informed of updates as values change. Specifically, this is the mechanism which enables the User Interface subsystem to maintain its workstation displays of Uplink Rate and Mode, Antenna Specification, Archive Filename, Archive State, and Channel Specification. TransmitCommand provides these values to the parameter server as they change. The parameter server, in turn, forwards the new values to those processes which have requested to be kept updated with parameter values.

EDOS : The TransmitCommand process meters out commands to EDOS for uplink to the spacecraft.

FopCommand: The FopCommand process forwards CLTUs to the TransmitCommand process for metering to EDOS.

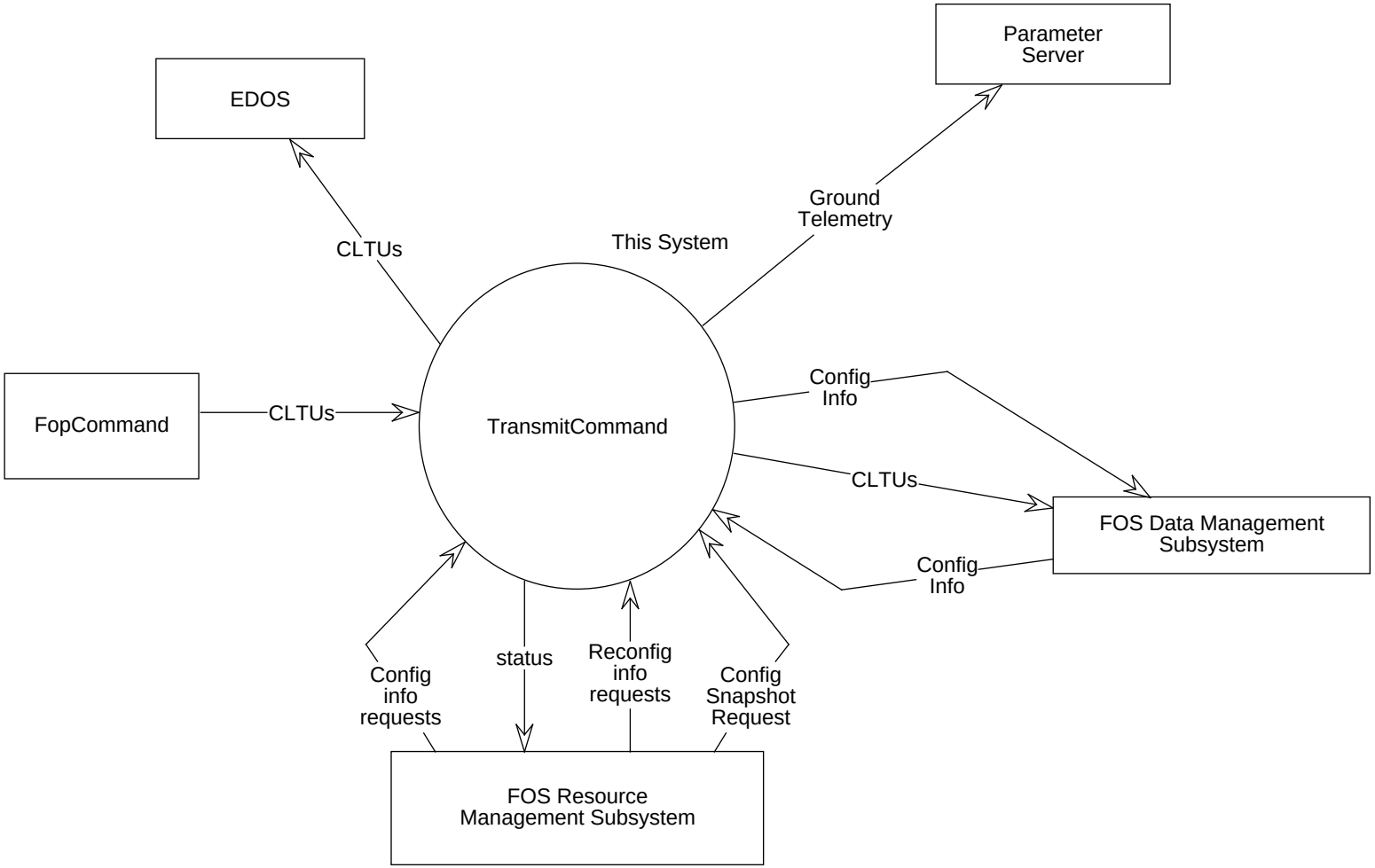

Figure 3.4.1-1. TransmitCommand Context Diagram

3.4.2 TransmitCommand Interfaces

***Table* 3.4.2. TransmitCommand Interfaces (1 of 2)**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Send CLTUs to EDOS	FoGnCmd Ground StationIF	Send binary commands to EDOS	CMD: Transmit	CMD: Transmit	once per command
Receives CLTUs	FcCm CCSDSFop IF	Receives CLTUs from the FopCommand task	CMD: Transmit	CMD: Fop	once per command
	FcGnTcCltu	Message class for a CLTU			
I/O	FoDsFile	Provides file access	DMS: FoDsFile Manager	CMD: Transmit	once per command
Provide Configuration Info	FoGnCmd Transmit RmsIF	Receive directives (other than commands)	CMD: Transmit	RMS: String Manager	< 2 x (twice per string configuration + once per pass)
	FoGnRms Config Msg	Contains config info for the TransmitCommand			
	FoGnRms Archive Msg	Allows for setting of the archive state (enable/disable)			
	FoGnRms Specify Channel Msg	Configure to accommodate the channel in the uplink path			
	FoGnRms Specify Antenna Msg	Configure to accommodate the antenna used in the uplink path			

***Table* 3.4.2. TransmitCommand Interfaces (2 of 2)**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
	FoGnRms Channel Antenna Msg	Configure for both channel & antenna used in uplink path			
	FoGnRms Primary ModeMsg	Configure task as either the primary or backup task			
	FoGnRms Shutdown Msg	Process will terminate itself, in an orderly manner			
	FoGnRms ReadSnap ShotMsg	Configure from a snapshot (nominally in backup mode)			
	FoGnRms SaveSnap ShotMsg	Take a snapshot (nominally in primary mode)			
	FoGnRms Transmit AckMsg	Acknowledges any of the above RMS messages			
Provides access to data values	FoGn Parameter Server	Distribution of updated values to other processes	Parameter Server	CMD: Transmit	~10x per r/t command
Event Logging	FdEvEvent Logger	Provides routing and archiving of events messages	DMS: FdEvEvent Archiver	CMD: Transmit	Once per r/t cmd, twice per load

3.4.3 TransmitCommand Object Model Description

The design scope for the TransmitCommand process Object Model (Figure 3.4.3-1) is the sending of commands to a single EOS spacecraft. Support for multiple spacecrafts / simulators (i.e., multiple logical strings) results in multiple instances of this model.

The FcCmTransmitController class controls the timing of the commands being sent. It is responsible for initialization of the TransmitCommand process. Commands, in CLTU format, are received from the FopCommand process via the FcCmCCSDSFopIF class. The FcCmTransmitController's role is to queue all received commands and to meter them out to the FoGnCmdGroundStationIF class.

The FcCmTransmitQueue class is a container class that contains instances of the FcCmTcCltu class. The FcCmTcCltu class contains the command in CLTU format and its size.

The RMS Interfaces for TransmitCommand are shown in Figure 3.4.3-2. The class FoGnRmsTransmitProxy represents the proxy of the TransmitCommand process to the RMS subsystem. The RMS subsystem uses this proxy to sends config info and directives to the TransmitCommand process. It also receives ack back from the TransmitCommand via the operation GetMessage of this proxy. The class FoGnCmdTransmitRmsIF is the interface class between the TransmitCommand process and the RMS subsystem. This class is used by the TransmitCommand to receive messages from and send ack message to RMS.

The messages that are sent from RMS to TransmitCommand process are represented by the classes: FoGnRmsConfigMsg, FoGnRmsArchiveMsg, FoGnRmsSpecifyChannelMsg, FoGnRmsSpecifyAntennaMsg, FoGnRmsChannelAntennaMsg, FoGnRmsPrimaryModeMsg, FoGnRmsShutdownMsg, FoGnRmsReadSnapshotMsg and FoGnRmsSaveSnapshotMsg.

The ack message from TransmitCommand to RMS is handled by the class FoGnCmdTransmitAckMsg.

The class FoGnCmdFopTransmitProxy (Figure 3.4.3-3) represents the proxy used by FopCommand process to send messages to TransmitCommand process. The message here is an instance of the FcGnTcCltu class. At the other end, TransmitCommand process uses FcCmCCSDSFopIF to receive messages from FopCommand.

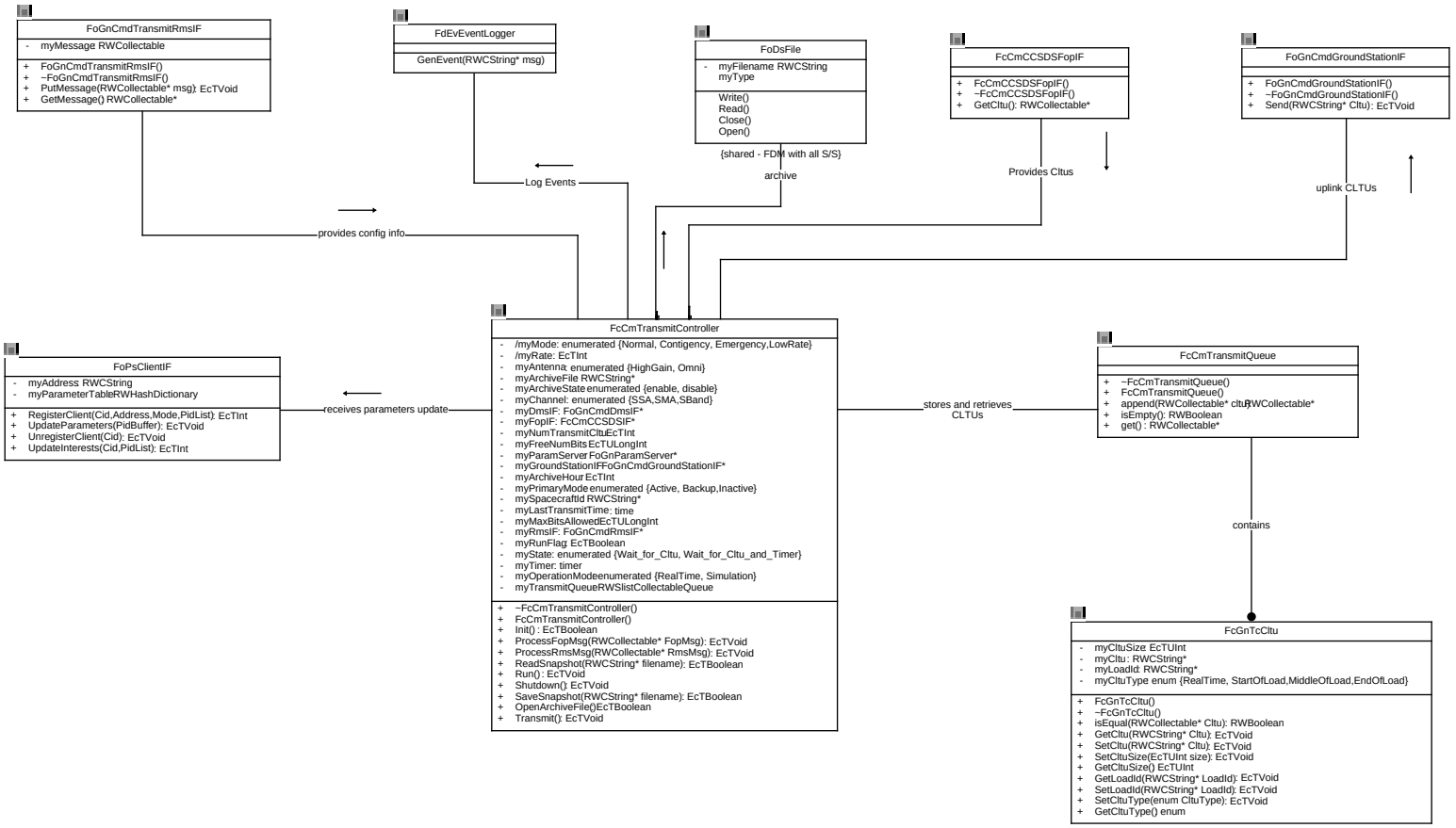

Figure 3.4.3-1. TransmitCommand Object Diagram

Figure 3.4.3-2. RMS / TransmitCommand I/F Object Diagram

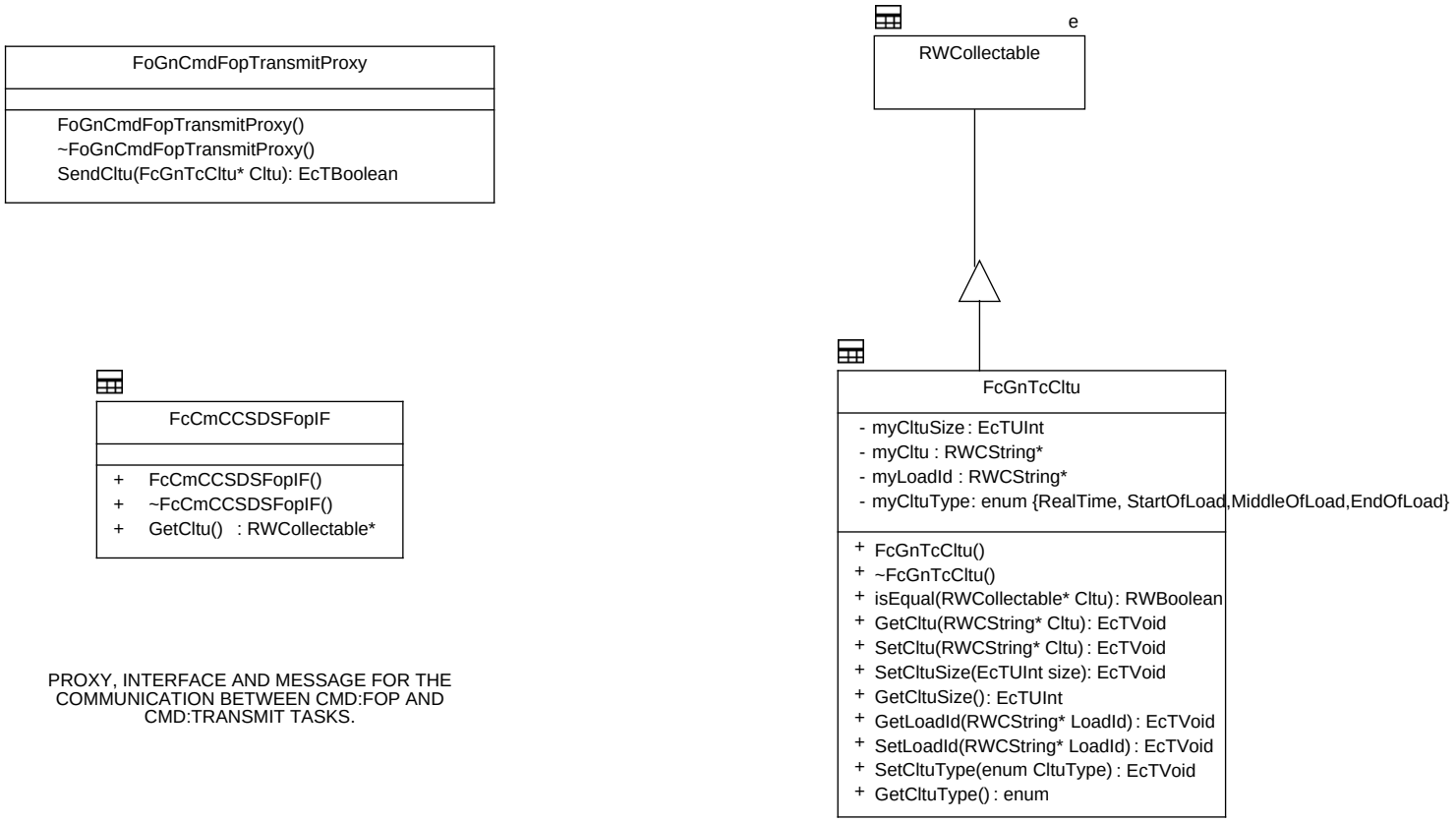

Figure 3.4.3-3. FopCommand / TransmitCommand I/F Object Diagram

3.4.4 TransmitCommand Dynamic Model Description

The following are the TransmitCommand scenarios which are defined in this section.

Real-Time Command Transmission

Real-Time Load Transmission

Additionally, a state diagram for the TransmitCommand controller class is included.

3.4.4.1 Real-Time Command Transmission Scenario

3.4.4.1.1 Real-Time Command Transmission Abstract

The purpose of the "Real-Time Command Transmission" scenario is to describe the process by which commands are metered to maximize bandwidth utilization.

Figure 3.4.4.1-1 is the event trace diagram which correspond to this scenario.

3.4.4.1.2 Real-Time Command Transmission Summary Information

Interfaces:

Data Management Subsystem

EDOS

FopCommand process

Stimulus:

The FopCommand process sends a CLTU to TransmitCommand.

Desired Response:

TransmitCommand forwards the CLTUs to EDOS, in a metered manner at a data rate of .125, 1, 2 or 10 kbps, depending on the uplink path.

Pre-Conditions:

The command queue (i.e., FcCmCommandQueue) is empty, and there are no CLTUs in transmission.

Post-Conditions:

The CLTUs have been successfully forwarded to EDOS.

3.4.4.1.3 Scenario Description

TransmitCommand receives a CLTU from FopCommand via FcCmCCSDSFopIF, and inserts it into the FcCmTransmitQueue. TransmitCommand checks the number of bits currently in transmission (by calculating the length of time since the last transmission, and the number of bits trasmitted at that time) to see if sending out the CLTU would result in the uplink rate being exceeded. Inasmuch as there are no commands in transmission, it is determined that the CLTU can be sent at this time without exceeding the uplink rate. The CLTU is removed from FcCmTransmitQueue and forwarded to EDOS via FoGnCmdGroundStationIF. The CLTU is archived via FoDsFile and an event message is logged via FdEvEventLogger. The queue is now empty, and the process is placed in a wait state until a message arrives.

TransmitCommand receives a second CLTU from FopCommand and inserts it into the

FcCmTransmitQueue. TransmitCommand checks the number of bits currently in transmission and determines that sending an additional CLTU would cause the uplink rate to be exceeded, and so it does not send the CLTU. A timer is set to expire at the time when the next CLTU can be sent out. FcCmTransmitController waits for either 1) the next message or 2) the timer to expire.

TransmitCommand receives a third CLTU from FopCommand and inserts it into the FcCmTransmitQueue. TransmitCommand returns to the wait state, waiting for either 1) the next message, or 2) the timer to expire. In this state, all of the incoming CLTUs are inserted into the queue. The timer expires. The next CLTU is removed from FcCmTransmitQueue and forwarded to EDOS via FoGnCmdGroundStationIF. The CLTU is archived via FoDsFile and an event message is logged via FdEvEventLogger.

The queue is not empty at this point. TransmitCommand checks the number of bits currently in transmission and determines that it cannot send another CLTU to EDOS at this time without exceeding the uplink rate. The timer is set again, and the process goes into the wait state waiting for either 1) the next CLTU from the FopCommand process or 2) the timer to expire. This process is repeated until the queue is once again empty.

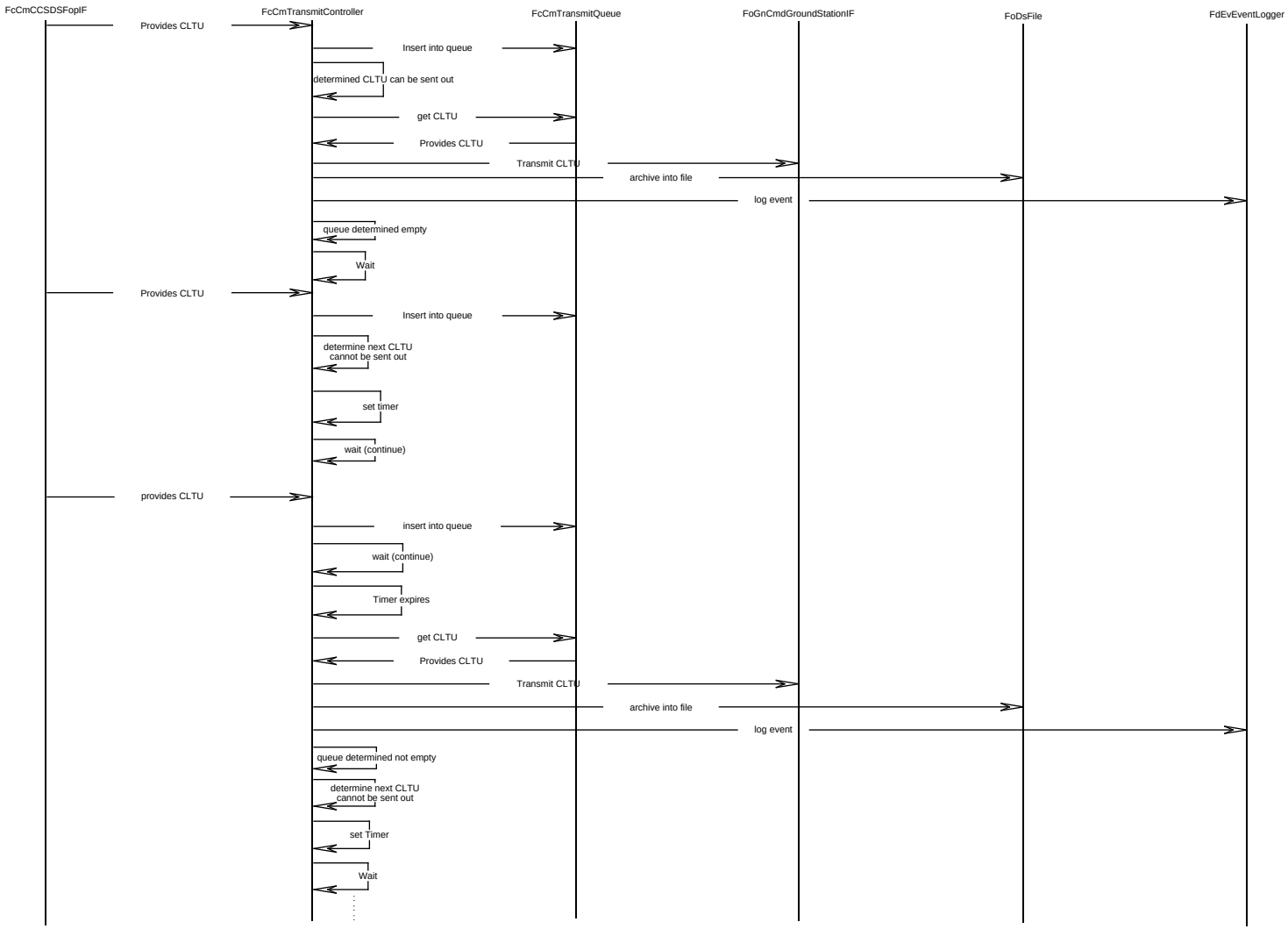

Figure 3.4.4.1-1. Real Time Command Transmission

3.4.4.2 Real-Time Load Transmission Scenario

3.4.4.2.1 Real-Time Load Transmission Abstract

The purpose of the "Real-Time Load Transmission" scenario is to describe the process by which CLTUs representing a load are metered to maximize bandwidth utilization.

Figure 3.4.4.2-1 is the event trace diagram which correspond to this scenario.

3.4.4.2.2 Real-Time Load Transmission Summary Information

Interfaces:

Data Management Subsystem

EDOS

FopCommand process

Stimulus:

The FopCommand process sends the first CLTU from a load to TransmitCommand.

Desired Response:

TransmitCommand forwards the CLTUs to EDOS, in a metered manner at a data rate of .125, 1, 2 or 10 kbps, depending on the uplink path.

Pre-Conditions:

The command queue (i.e., FcCmCommandQueue) is empty, and there are no CLTUs in transmission.

Post-Conditions:

The CLTUs have been successfully forwarded to EDOS.

3.4.4.2.3 Scenario Description

TransmitCommand receives the first CLTU in a load sequence from FopCommand via FcCmCCSDSFopIF, and inserts it into the FcCmTransmitQueue. TransmitCommand checks the number of bits currently in transmission (by calculating the length of time since the last transmission, and the number of bits transmitted at that time) to see if sending out the CLTU would result in the uplink rate being exceeded. Inasmuch as there are no commands in transmission, it is determined that the CLTU can be sent at this time without exceeding the uplink rate. The CLTU is removed from FcCmTransmitQueue and forwarded to EDOS via FoGnCmdGroundStationIF. The CLTU is archived via FoDsFile and an event message marking the start of the uplink of the load is logged via FdEvEventLogger. The queue is now empty, and the process is placed in a wait state until a message arrives.

TransmitCommand receives a second CLTU from FopCommand and inserts it into the FcCmTransmitQueue. TransmitCommand checks the number of bits currently in transmission and determines that sending an additional CLTU would cause the uplink rate to be exceeded, and so it does not send the CLTU. A timer is set to expire at the time when the next CLTU can be sent out. FcCmTransmitController waits for either 1) the next message or 2) the timer to expire.

TransmitCommand receives a third CLTU from FopCommand and inserts it into the FcCmTransmitQueue. TransmitCommand returns to the wait state, waiting for either 1) the next message, or 2) the timer to expire. In this state, all of the incoming CLTUs are inserted into the queue. The timer expires. The next CLTU is removed from FcCmTransmitQueue and forwarded to EDOS via FoGnCmdGroundStationIF. The CLTU is archived via FoDsFile. The Controller recognizes that the CLTU is part of the load, and as such, does not log an event message for this CLTU.

The queue is not empty at this point. TransmitCommand checks the number of bits currently in transmission and determines that it cannot send another CLTU to EDOS at this time without exceeding the uplink rate. The timer is set again, and the process goes into the wait state waiting for either 1) the next CLTU from the FopCommand process or 2) the timer to expire. This process is repeated for all CLTUs, except for the last CLTU in the load.

Eventually, the timer expires, and the next CLTU is removed from FcCmTransmitQueue and forwarded to EDOS via FoGnCmdGroundStationIF. The CLTU is archived via FoDsFile. The Controller recognizes that the CLTU is the last CLTU for the load, and an event message marking the end of the uplink of the load is logged via FdEvEventLogger.

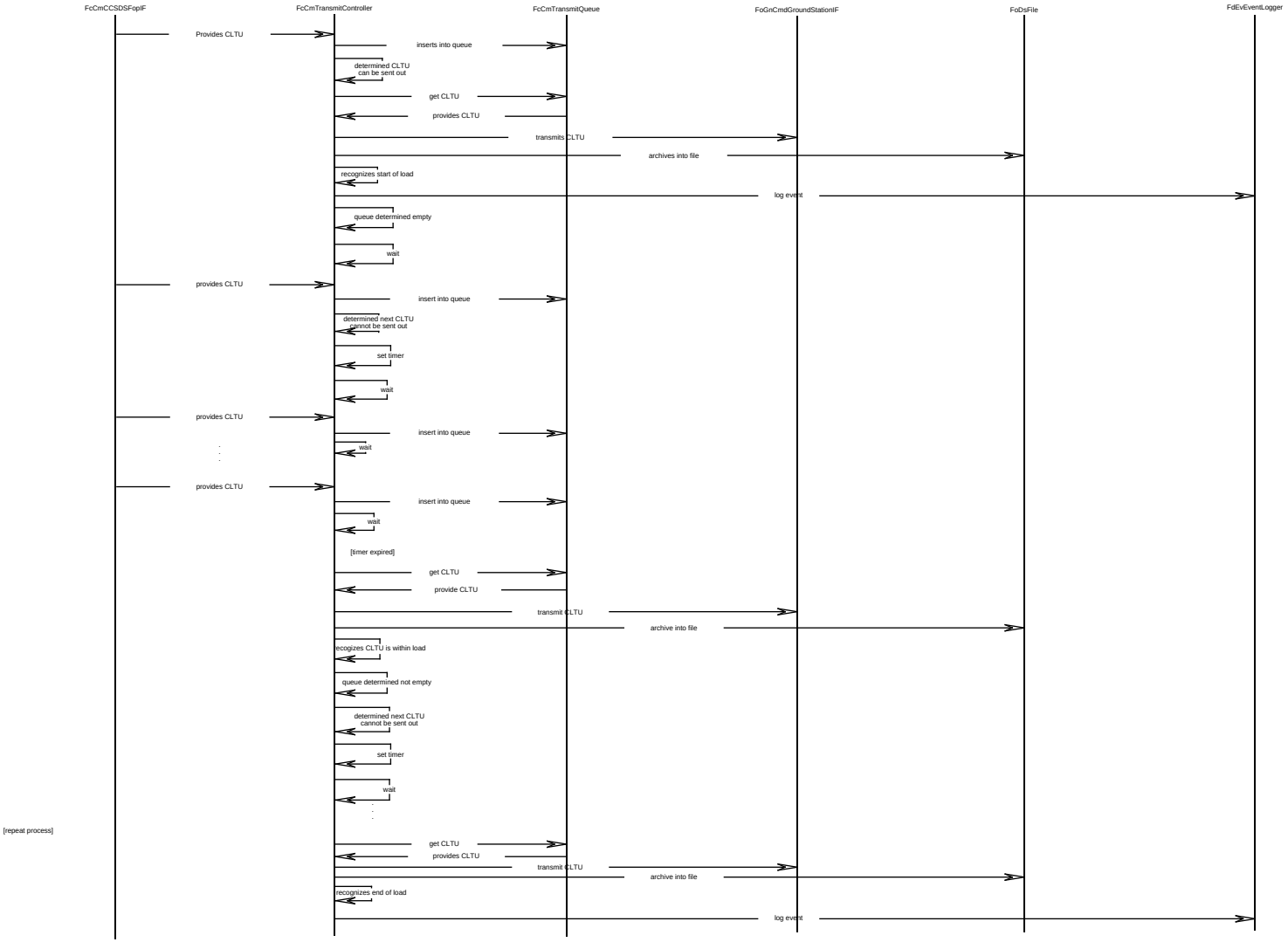

Figure 3.4.4.2-1. Real Time Load Command Transmission

3.4.4.3 FcCmTransmitController State Diagram Description

Once initialized, the FcCmTransmitController object enters Wait_for_Cltu state. In this state, it can receive and process configuration and shutdown messages from RMS, and CLTUs from CMD:Fop process. It only leaves this state for another state if it receives either a shutdown message from RMS or CLTUs from CMD:Fop.

Upon receiving CLTUs from CMD:Fop in the Wait_for_Cltu state, the controller object inserts all incoming CLTUs into FcCmTransmitQueue object. It then checks to see if it can forward any CLTU to EDOS without exceeding the uplink rate. If it can, then the controller will leave this state for the Transmit state. If it cannot, it will calculate the time that the next CLTU in the queue may be sent. The controller then sets the timer accordingly and goes to the Wait_for_Cltu_and_Timer state.

In the Transmit state, the controller sends as many CLTUs to EDOS as possible, without exceeding bandwidth. Then, if the queue is empty, it returns to the Wait_for_Cltu state. Otherwise, it calculates the time that it needs to wait before sending the next CLTU. It then sets the timer and goes to the Wait_for_Cltu_and_Timer state.

In the Wait_for_Cltu_and_Timer state, the controller object receives CLTUs and messages from other subsystems just like when it is in the Wait_for_Cltu state. The only difference is that, when receiving CLTUs from CMD:Fop, it enqueues the CLTUs and waits for the timer to expire before it can enter the Tranmit state.

In either the Wait_for_Cltu or Wait_for_Cltu_and_Timer state, upon receiving the shutdown message from RMS, the controller object will enter the shutdown state, then exit.

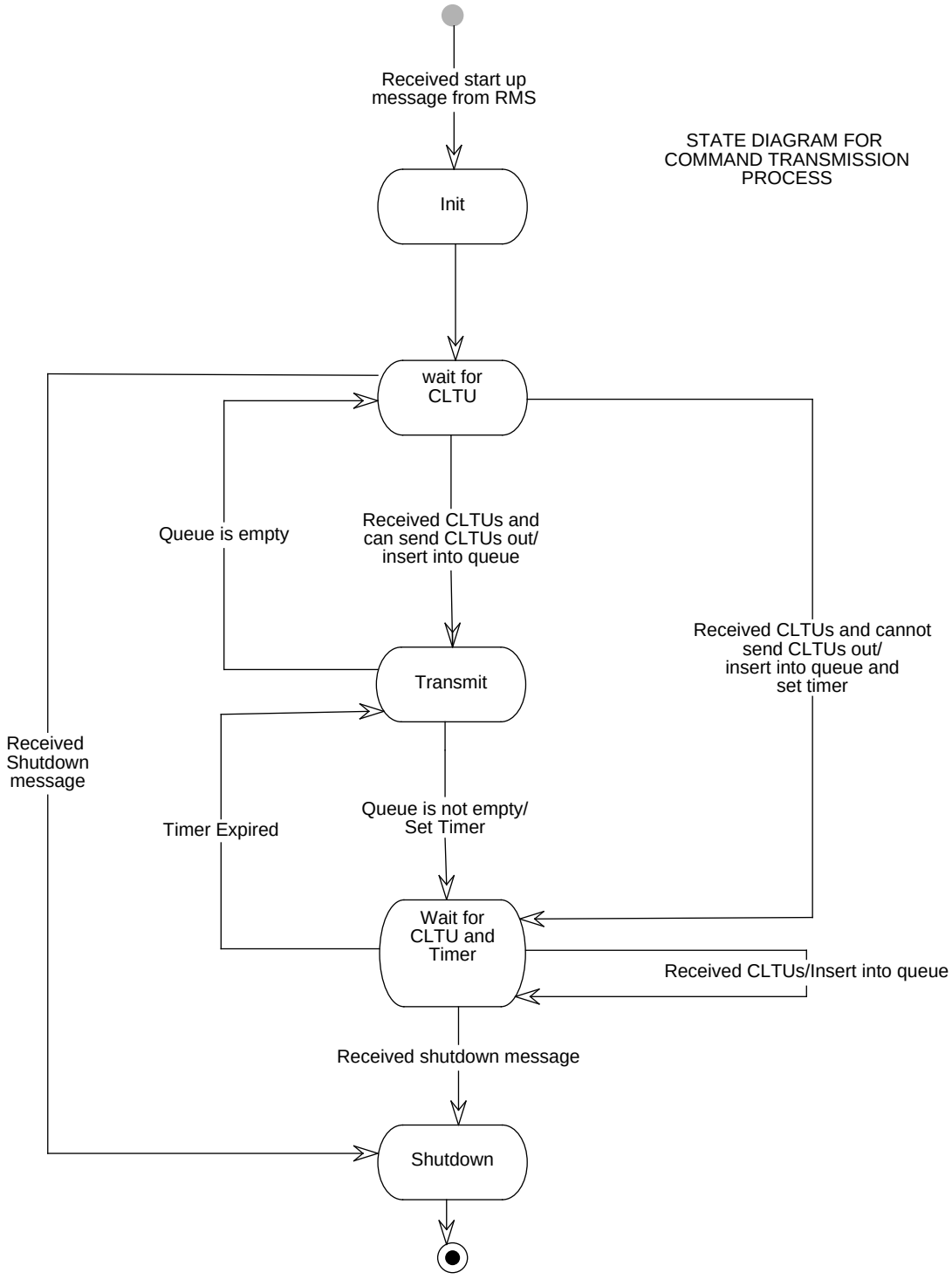

Figure 3.4.4.3-1. FcCmTransmitController state diagram

3.4.5 TransmitCommand Data Dictionary

FcCmCCSDSFopIF

class **FcCmCCSDSFopIF**

This class handles the exchange of information between the Transmit task and the Fop Command task.

Public Construction

FcCmCCSDSFopIF()

This member function is the default constructor

~FcCmCCSDSFopIF()

This member function is the destructor

Public Functions

RWCollectable* **GetCltu**(void)

This member function returns the pointer to the Cltu forwarded from Fop.

FcCmTransmitController

class **FcCmTransmitController**

This class is the controller class for the Command Transmit process. It is responsible for receiving CLTUs from Command Fop process and uplink these CLTUs out at a specified transmission rate.

Public Construction

FcCmTransmitController()

This member function is the default constructor for the Transmit Controller.

~FcCmTransmitController()

This member function is the destructor for the Transmit Controller.

Public Functions

EcTBoolean **Init**(void)

This member function initializes the Transmit Controller.

EcTBoolean **OpenArchiveFile**(void)

This member function opens the file for archiving CLTUs. It uses the spacecraft ID, the operational mode (real time or simulation) and time stamp (year,month,day,hour) for the archive file name.

EcTVoid **ProcessFopMsg**(FopMsg)

This member function accepts CLTUs from the command Fop task.

```
*  If the current state is Wait_for_Cltu, it then determines if it
```

can send out a CLTU. If it can, it calls the Transmit function. Otherwise, it sets the timer to go off when a CLTU can be sent out.

```
*  If the current state is Wait_for_Cltu_and_Timer, it enqueues
all received CLTUs into the transmit queue.
```

EcTVoid **ProcessRmsMsg**(RmsMsg)

This member function processes directives from RMS process.

EcTBoolean **ReadSnapshot**(RWCString* filename)

This member function reads out the antenna, channel and archive state from the config file.

`EcTVoid` **`Run`**`(void)`

This member function waits for and receives messages from Fop or Rms.

`EcTBoolean` **`SaveSnapshot`**`(RWCString* name)`

This member function writes out the antenna, channel and archive state to the snapshot file.

`EcTVoid` **`Shutdown`**`(void)`

This member function does the cleanup job before the termination of the transmit process.

`EcTVoid` **`Transmit`**`(void)`

This member function sends out one CLTU at a time while it can. When it cannot send out any more CLTU and there are still more CLTUs to be sent out, it will set the timer to go off at the time when it can send out the next CLTU .

Private Data

`enumerated` **`_myMode`**

myMode

This member variable is derived from myAntenna and myChannel. It contains the mode of the current transmission (e.g. normal, contingency, emergency).

`EcTInt` **`_myRate`**

myRate

This member variable is derived from myAntenna and myChannel. It contains the current transmission rate.

`enumerated` **`myAntenna`**

This member variable contains the current antenna used in spacecraft receiving commands.

`RWCString*` **`myArchiveFile`**

This member variable points to the file used for archiving uplinked CLTUs.

`EcTInt` **`myArchiveHour`**

This member variable contains the hour when the archive file was opened.

`enumerated` **`myArchiveState`**

This member variable contains the state of archiving, i.e. On or Off.

`enumerated` **`myChannel`**

This member variable contains the uplink channel, e.g. SSA (S band single access), SMA (S band multiple access).

`FoGnCmdDmsIF*` **`myDmsIF`**

This member variable points to the interface to DMS .

`FcCmCCSDSIF*` **`myFopIF`**

This member variable points to the interface to Cmd:Fop task.

`EcTULongInt` **`myFreeNumBits`**

This member variable contains the number of bits that can be transmitted at the current time.

`FoGnCmdGroundStationIF*` **`myGroundStationIF`**

This member variable points to the interface to EDOS .

`time` **`myLastTransmitTime`**

This member variable contains the time of the last transmission.

`EcTULongInt` **`myMaxBitsAllowed`**

This member variable contains the size of the largest CLTU.

`EcTInt` **`myNumTransmitCltu`**

This member variable contains the number of Cltu in a load

`enumerated` **`myOperationMode`**

This member variable contains the current operational mode, e.g. real-time or simulation:

`FoGnParamServer*` **`myParamServer`**

This member variable points to the parameter server

`enumerated` **`myPrimaryMode`**

This member variable contains the state of the process, e.g. primary ,backup or inactive.

`FoGnCmdRmsIF*` **`myRmsIF`**

This member variable points to the interface to RMS.

`EcTBoolean` **`myRunFlag`**

This member variable contains the indicator telling the transmit controller object whether or not to stop the task.

`RWCString*` **`mySpacecraftId`**

This member variable contains the ID of the spacecraft.

`enumerated` **`myState`**

This member variable contains the current state of the transmit controller object (e.g. Wait_for_Cltu, Wait_for_Cltu_and_Timer).

`timer` **`myTimer`**

This member variable contains the id of the timer.

`RWSlistCollectableQueue*` **`myTransmitQueue`**

This member variable points to the transmit queue.

FcCmTransmitQueue

class **FcCmTransmitQueue**

This class is derived from Rogue Wave RWSlistCollectableQueue class. It is a container class that contains CLTUs to be uplinked.

Base Classes

`public` **`RWSlistCollectableQueue`**

Public Construction

`FcCmTransmitQueue``()`

This member function is the default constructor.

`~FcCmTransmitQueue``()`

This member function is the default destructor.

Public Functions

`EcTBoolean` **`append`**`(RWCollectable* cltu)`

This member function append one CLTU into the queue.

`RWCollectable*` **`get`**`(void)`

This member function get the next CLTU from the queue.

`RWBoolean` **`isEmpty`**`(void)`

This member function checks to see if the queue is empty.

FcGnTcCltu

class **FcGnTcCltu**

This class contains the message passed from Fop to Transmit task.

Public Construction

FcGnTcCltu(void)

This member function is the default constructor

~FcGnTcCltu(void)

This member function is the destructor.

Public Functions

GetCltuType(void)

This member function returns myCltuType

EcTVoid **GetCltu**(RWCString* Cltu)

This member function returns myCltu

EcTUInt **GetCltuSize**(void)

This member function returns myCltuSize

EcTVoid **GetLoadId**(RWCString* LoadId)

This member function returns the LoadId

EcTVoid **SetCltu**(RWCString* Cltu)

This member function sets myCltu attribute.

EcTVoid **SetCltuSize**(EcTUInt size)

This member function set myCltuSize

EcTVoid **SetCltuType**(CltuType)

This member function sets myCltuType

EcTVoid **SetLoadId**(RWCString* LoadId)

This member function sets the loadId

RWBoolean **isEqual**(RWCollectable* Cltu)

This member function is required by Rogue Wave but is not used here.

Private Data

RWCString* **myCltu**

This member variable contains the CLTU

EcTUInt **myCltuSize**

This member variable contains the size of the CLTU

enum **myCltuType**

Private Types

enum

This member variable contains the type of the CLTU (real-time, StartOfLoad, MiddleOfLoad or EndOfLoad)

Enumerators

EndOfLoad
MiddleOfLoad
RealTime
StartOfLoad

FoGnCmdFopTransmitProxy

class **FoGnCmdFopTransmitProxy**

This class implements the Transmit Proxy for the Fop task.

Public Construction

FoGnCmdFopTransmitProxy()

This member function is the default constructor

~FoGnCmdFopTransmitProxy()

This member function is the destructor

Public Functions

EcTBoolean **SendCltu**(FcGnTcCltu* Cltu)

Transmit This member function send the Cltu to Transmit task

FoGnCmdGroundStationIF

class **FoGnCmdGroundStationIF**

Public Construction

FoGnCmdGroundStationIF()

This member function is the default constructor

~FoGnCmdGroundStationIF()

This member function is the destructor.

Public Functions

EcTVoid **Send**(RWCString* Cltu)

This member function takes a CLTU and sends it out.

FoGnCmdTransmitAckMsg

class **FoGnCmdTransmitAckMsg**

This class implements the ack message sent from Cmd:Transmit task to Rms subsystem.

Public Construction

FoGnCmdTransmitAckMsg(void)

This is the default constructor.

~FoGnCmdTransmitAckMsg(void)

This is the destructor.

Public Functions

EcTBoolean **GetStatus**(void)

This member function returns myAckStatus

`EcTVoid` **`SetStatus`**`(EcTBoolean Status)`

This member function sets myAckStatus to status.

Private Data

`EcTBoolean` **`myAckStatus`**

This member variable contains the status.

FoGnCmdTransmitRmsIF

class `FoGnCmdTransmitRmsIF`

This class implements the interface from Cmd:Transmit task to Rms subsystem

Public Construction

`FoGnCmdTransmitRmsIF``(void)`

This is the default constructor.

`~FoGnCmdTransmitRmsIF``(void)`

This is the destructor.

Public Functions

`RWCollectable*` **`GetMessage`**`()`

This member function returns the address to the object message forwarded to this interface.

`EcTVoid` **`PutMessage`**`(RWCollectable* msg)`

This member function sends a RWCollectable object out.

Private Data

`RWCollectable` **`myMessage`**

The message that this interface holds

FoGnRmsArchiveMsg

class `FoGnRmsArchiveMsg`

This class contains the Archive message.

Public Construction

`FoGnRmsArchiveMsg``(void)`

This member function is the default constructor

`~FoGnRmsArchiveMsg``(void)`

This member function is the destructor.

Public Functions

`GetArchiveState(void)`

This member function returns the value of the attribute

`EcTVoid` **`SetArchiveState`**`(NewState)`

This member function set the attribute to NewState

Private Data

enum **myArchiveState**

Private Types

enum

This member variable contains the archive state

Enumerators

disable
enable

FoGnRmsChannelAntennaMsg

class **FoGnRmsChannelAntennaMsg**

This class contains the message to update Channel and Antenna

Public Construction

FoGnRmsChannelAntennaMsg(void)

This member function is the default constructor

~FoGnRmsChannelAntennaMsg(void)

This member function is the destructor

Public Functions

EcTVoid **GetChannelAntenna**(NewChannel, NewAntenna)

This member function returns the values of attributes.

EcTVoid **SetChannelAntenna**(NewChannel, NewAntenna)

This member function set the attributes to new values

Private Data

enum **myAntenna**

enum **myChannel**

Private Types

enum

This member variable contains the name of the antenna

Enumerators

HighGain
Omni

enum

This member variable contains the name of the channel

Enumerators

SBand
SMA
SSA

FoGnRmsConfigMsg

class **FoGnRmsConfigMsg**

This class contains the config message that Rms send to Cmd:Transmit.

Public Construction

FoGnRmsConfigMsg(void)

This member function is the default constructor.

~FoGnRmsConfigMsg(void)

This member function is the destructor.

Public Functions

EcTVoid **GetConfig**(RWCString* SpacecraftId, RWCString* DbId, enum(Active))

This member function returns all the config attributes.

Private Data

FoGnCmdFop* **myCmdFopAddr**

myFopAddr

This member variable contains the address of the Fop Task.

RWCString* **myDbId**

This member variable contains the database ID

enum **myOperationMode**

FoGnParamServer* **myParamServer**

This member variable contains the address of the parameter server.

enum **myPrimaryMode**

RWCString* **mySpacecraftId**

This member variable contains the Id of the spacecraft.

Private Types

enum

This member variable contains the operational mode of the process.

Enumerators

RealTime
Simulation

enum

This member variable contains the mode of the process.

Enumerators

Active
Backup

FoGnRmsPrimaryModeMsg

class **FoGnRmsPrimaryModeMsg**

This class contains the new Primary Mode.

Public Construction

FoGnRmsPrimaryModeMsg(void)

This member function is the default constructor.

~FoGnRmsPrimaryModeMsg(void)

This member function is the destructor.

Public Functions

GetPrimaryMode(void)

This member function returns the value of the attribute

EcTVoid **SetPrimaryMode**(NewMode)

This member function sets the value of the attribute

Private Data

enum **myPrimaryMode**

Private Types

enum

This member variable contains the mode of operation

Enumerators

active
backup

FoGnRmsReadSnapshotMsg

class **FoGnRmsReadSnapshotMsg**

This class is derived from FoGnRmsSnapshotMsg; it contains the name of the snapshot file.

Base Classes

public **FoGnRmsSnapshotMsg**

Public Construction

FoGnRmsReadSnapshotMsg(void)

FoGnRmsReadSnapshopMsg

This member function is the default constructor

~FoGnRmsReadSnapshotMsg(void)

This member function is the destructor.

FoGnRmsSaveSnapshotMsg

class **FoGnRmsSaveSnapshotMsg**

This class is derived from FoGnRmsSnapshotMsg; it contains the filename to save the snapshot.

Base Classes

public **FoGnRmsSnapshotMsg**

Public Construction

FoGnRmsSaveSnapshotMsg(void)

This member function is the default constructor

~FoGnRmsSaveSnapshotMsg(void)

This member function is the destructor

FoGnRmsShutdownMsg

class **FoGnRmsShutdownMsg**

This class contains the shutdown message via its type (isA relation)

Public Construction

FoGnRmsShutdownMsg(void)

This member function is the default constructor

~FoGnRmsShutdownMsg(void)

This member function is the destructor

FoGnRmsSnapshotMsg

class **FoGnRmsSnapshotMsg**

This is the base class for FoGnRmsReadSnapshotMsg and FoGnRmsSaveSnapshoMsg classes.

Public Construction

FoGnRmsSnapshotMsg(void)

FoGnRmsSnapshopMsg

This member function is the default constructor

~FoGnRmsSnapshotMsg(void)

This member function is the destructor.

Public Functions

EcTVoid **GetFilename**(RWCString* filename)

This member function returns the value of the attribute.

EcTVoid **SetFilename**(RWCString* filename)

This member function sets the value of the attribute

Private Data

RWCString* **myFilename**

This member variable contains the name of the snapshot file

FoGnRmsSpecifyAntennaMsg

class **FoGnRmsSpecifyAntennaMsg**

This class contains the Specify Antenna message.

Public Construction

FoGnRmsSpecifyAntennaMsg(void)

FoGnRMsSpecifyAntennaMsg

This member function is the default constructor.

~FoGnRmsSpecifyAntennaMsg(void)

~FoGnRMsSpecifyAntennaMsg

This member function is the destructor.

Public Functions

GetAntenna(void)

This member function returns the value of the attribute.

EcTVoid **SetAntenna**(NewAntenna)

This member function sets the value of the attribute.

Private Data

enum **myAntenna**

Private Types

enum

This member variable contains the name of the antenna

Enumerators

HighGain
Omni

FoGnRmsSpecifyChannelMsg

class **FoGnRmsSpecifyChannelMsg**

This class contains the Specify Channel message.

Public Construction

FoGnRmsSpecifyChannelMsg(void)

This member function is the default constructor.

~FoGnRmsSpecifyChannelMsg(void)

This member function is the destructor.

Public Functions

GetChannel(void)

This member function returns the value of the attribute.

EcTVoid **SetChannel**(NewChannel)

This member function sets the value of the attribute.

Private Data

`enum` **`myChannel`**

Private Types

`enum`

This member variable contains the channel name.

Enumerators

SBand
SMA
SSA

FoGnRmsTransmitProxy

`class` **`FoGnRmsTransmitProxy`**

This class is the class for Transmit Proxy for RMS task

Public Construction

`FoGnRmsTransmitProxy``(void)`

This member function is the default constructor

`~FoGnRmsTransmitProxy``(void)`

This member function is the destructor

Public Functions

`EcTBoolean` **`Archive`**`(ArchiveState)`

This member function creates an FoGnRmsArchiveMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

`EcTBoolean` **`Config`**`(SpacecraftId, DbId, PrimaryMode, OpMode, ParamServer, CmdFop)`

This member function creates an FoGnRmsConfigMsg object and sets its attributes to those of the arguments, then sends it to Cmd:Transmit

`EcTBoolean` **`ConfigurationSnapshotRequest`**`(RWCString* filename)`

This member function creates an FoGnRmsSaveSnapshotMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

`RWCollectable*` **`GetMessage`**`()`

Returns the pointer to the message

`EcTBoolean` **`ReadConfigurationSnapshot`**`(RWCString* filename)`

This member function creates an FoGnRmsReadSnapshotMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

`EcTBoolean` **`SelectPrimaryMode`**`(NewPrimaryMode)`

This member function creates an FoGnRmsPrimaryModeMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

`EcTBoolean` **`Shutdown`**`(void)`

This member function creates an FoGnRmsShutdownMsg object and sends it to the Cmd:Transmit task.

`EcTBoolean` **`SpecifyAntenna`**`(NewAntenna)`

This member function creates an FoGnRmsSpecifyAntennaMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

EcTBoolean **SpecifyChannel**(NewChannel)

This member function creates an FoGnRmsSpecifyChannelMsg object and sets its attribute to that of the argument, then sends it to Cmd:Transmit task

EcTBoolean **SpecifyChannelAndAntenna**(NewChannel, NewAntenna)

This member function creates an FoGnRmsChannelAntennaMsg object and sets its attributes to those of the arguments, then sends it to Cmd:Transmit task

Public Types

enum **ArchiveState**

Enumerators

disable
enable

enum **NewAntenna**

Enumerators

HighGain
Omni

enum **NewChannel**

Enumerators

S
SMA
SSA

enum **NewPrimaryMode**

Enumerators

Active
Backup

Private Data

RWCollectable **myMessage**

This member variable contains the received message .

Abbreviations and Acronyms

ACL	Access Control List
AD	Acceptance Check/TC Data
AGS	ASTER Ground System
AM	Morning (ante meridian) -- see EOS AM
Ao	Availability
APID	Application Identifier
ARAM	Automated Reliability/Availability/Maintainability
ASTER	Advanced Spaceborne Thermal Emission and Reflection Radiometer (formerly ITIR)
ATC	Absolute Time Command
BAP	Baseline Activity Profile
BC	Bypass check/Control Commands
BD	Bypass check/TC Data (Expedited Service)
BDU	Bus Data Unit
bps	bits per second
CAC	Command Activity Controller
CCB	Change Control Board
CCSDS	Consultative Committee for Space Data Systems
CCTI	Control Center Technology Interchange
CD-ROM	Compact Disk-Read Only Memory
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CERES	Clouds and Earth's Radiant Energy System
CI	Configuration item
CIL	Critical Items List
CLCW	Command Link Control Words
CLTU	Command Link Transmission Unit
CMD	Command subsystem
CMS	Command Management Subsystem
CODA	Customer Operations Data Accounting
COP	Command Operations Procedure
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit

CRC	Cyclic Redundancy Code
CSCI	Computer software configuration item
CSMS	Communications and Systems Management Segment
CSS	Communications Subsystem (CSMS)
CSTOL	Customer System Test and Operations Language
CTIU	Command and Telemetry Interface Unit (AM-1)
DAAC	Distributed Active Archive Center
DAR	Data Acquisition Request
DAS	Detailed Activity Schedule
DAT	Digital Audio Tape
DB	Data Base
DBA	Database Administrator
DBMS	Database Management System
DCE	Distributed Computing Environment
DCP	Default Configuration Procedure
DEC	Digital Equipment Corporation
DES	Data Encryption Standard
DFCD	Data Format Control Document
DID	Data Item Description
DMS	Data Management Subsystem
DOD	Digital Optical Data
DoD	Department of Defense
DS	Data Server
DSN	Deep Space Network
DSS	Decision Support System
e-mail	electronic mail
Ecom	EOS Communication
ECS	EOSDIS Core System
EDOS	EOS Data and Operations System
EDU	EDOS Data Unit
EGS	EOS Ground System
EOC	Earth Observation Center (Japan); EOS Operations Center (ECS)
EOD	Entering Orbital Day
EON	Entering Orbital Night
EOS	Earth Observing System

EOSDIS	EOS Data and Information System
EPS	Encapsulated Postscript
ESH	EDOS Service Header
ESN	EOSDIS Science Network
ETS	EOS Test System
EU	Engineering Unit
EUVE	Extreme Ultra Violet Explorer
FAS	FOS Analysis Subsystem
FAST	Fast Auroral Snapshot Explorer
FDDI	Fiber Distributed Data Interface
FDF	Flight Dynamics Facility
FDIR	Fault Detection and Isolation Recovery
FDM	FOS Data Management Subsystem
FMEA	Failure Modes and Effects Analyses
FOP	Frame Operations Procedure
FORMATS	FDF Orbital and Mission Aids Transformation System
FOS	Flight Operations Segment
FOT	Flight Operations Team
FOV	Field-Of-View
FPS	Fast Packet Switch
FRM	FOS Resource Management Subsystem
FSE	FOT S/C Evolutions
FTL	FOS Telemetry Subsystem
FUI	FOS User Interface
GB	Gigabytes
GCM	Global Circulation Model
GCMR	Global Circulation Model Request
GIMTACS	GOES I-M Telemetry and Command System
GMT	Greenwich Mean Time
GN	Ground Network
GOES	Geostationary Operational Environmental Satellite
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
H&S	Health and Safety
H/K	Housekeeking
HST	Hubble Space Telescope

I/F	Interface
I/O	Input/Output
ICC	Instrument Control Center
ICD	Interface Control Document
ID	Identifier
IDB	Instrument Database
IDR	Incremental Design Review
IEEE	Institute of Electrical and Electronics Engineers
IOT	Instrument Operations Team
IP	International Partners
IP-ICC	International Partners-Instrument Control Center
IPs	International Partners
IRD	Interface requirements document
ISDN	Integrated Systems Digital Network
ISOLAN	Isolated Local Area Network
ISR	Input Schedule Request
IST	Instrument Support Terminal
IST	Instrument Support Toolkit
IWG	Investigator Working Group
JPL	Jet Propulsion Laboratory
Kbps	Kilobits per second
LAN	Local Area Network
LaRC	Langley Research Center
LASP	Laboratory for Atmospheric Studies Project
LEO	Low Earth Orbit
LOS	Loss of Signal
LSM	Local System Manager
LTIP	Long-Term Instrument Plan
LTSP	Long-Term Science Plan
MAC	Medium Access Control; Message Authentication Code
MB	Megabytes
MBONE	Multicast Backbone
Mbps	Megabits per second
MDT	Mean Down Time
MIB	Management Information Base

MISR	Multi-angle Imaging Spectro-Radiometer
MMM	Minimum, Maximum, and Mean
MO&DSD	Mission Operations and Data Systems Directorate (GSFC Code 500)
MODIS	Moderate resolution Imaging Spectrometer
MOPITT	Measurements Of Pollution In The Troposphere
MSS	Management Subsystem
MTPE	Mission to Planet Earth
NASA	National Aeronautics and Space Administration
Nascom	NASA Communications Network
NASDA	National Space Development Agency (Japan)
NCAR	National Center for Atmospheric Research
NCC	Network Control Center
NEC	North Equator Crossing
NFS	Network File System
NOAA	National Oceanic and Atmospheric Administration
NSI	NASA Science Internet
NTT	Nippon Telephone and Telegraph
OASIS	Operations and Science Instrument Support
ODB	Operational Database
ODM	Operational Data Message
OMT	Object Model Technique
OO	Object Oriented
OOD	Object Oriented Design
OpLAN	Operational LAN
OSI	Open System Interconnect
PACS	Polar Acquisition and Command System
PAS	Planning and Scheduling
PDB	Project Data Base
PDF	Publisher's Display Format
PDL	Program Design Language
PDR	Preliminary Design Review
PI	Principal Investigator
PI/TL	Principal Investigator/Team Leader
PID	Parameter ID
PIN	Password Identification Number
POLAR	Polar Plasma Laboratory

POP	Polar-Orbiting Platform
POSIX	Portable Operating System for Computing Environments
PSAT	Predicted Site Acquisition Table
PSTOL	PORTS System Test and Operation Language
Q/L	Quick Look
R/T	Real-Time
RAID	Redundant Array of Inexpensive Disks
RCM	Real-Time Contact Management
RDBMS	Relational Database Management System
RMA	Reliability, Maintainability, Availability
RMON	Remote Monitoring
RMS	Resource Management Subsystem
RPC	Remote Processing Computer
RTCS	Relative Time Command Sequence
RTS	Relative Time Sequence; Real-Time Server
S/C	Spacecraft
SAR	Schedule Add Requests
SCC	Spacecraft Controls Computer
SCF	Science Computing Facility
SCL	Spacecraft Command Language
SDF	Software Development Facility
SDPS	Science Data Processing Segment
SDVF	Software Development and Validation Facility
SEAS	Systems, Engineering, and Analysis Support
SEC	South Equator Crossing
SLAN	Support LAN
SMA	S-band Multiple Access
SMC	Service Management Center
SN	Space Network
SNMP	System Network Mgt Protocol
SQL	Structured Query Language
SSA	S-band Single Access
SSIM	Spacecraft Simulator
SSR	Solid State Recorder
STOL	System Test and Operations Language

T&C	Telemetry and Command
TAE	Transportable Applications Environment
TBD	To Be Determined
TBR	To Be Replaced/Resolved/Reviewed
TCP	Transmission Control Protocol
TD	Target Day
TDM	Time Division Multiplex
TDRS	Tracking and Data Relay Satellite
TDRSS	Tracking and Data Relay Satellite System
TIROS	Television Infrared Operational Satellite
TL	Team Leader
TLM	Telemetry subsystem
TMON	Telemetry Monitor
TOO	Target Of Opportunity
TOPEX	Topography Ocean Experiment
TPOCC	Transportable Payload Operations Control Center
TRMM	Tropical Rainfall Measuring Mission
TRUST	TDRSS Resource User Support Terminal
TSS	TDRSS Service Session
TSTOL	TRMM System Test and Operations Language
TW	Target Week
U.S.	United States
UAV	User Antenna View
UI	User Interface
UPS	User Planning System
US	User Station
UTC	Universal Time Code; Universal Time Coordinated
VAX	Virtual Extended Address
VMS	Virtual Memory System
W/S	Workstation
WAN	Wide Area Network
WOTS	Wallops Orbital Tracking Station
XTE	X-Ray Timing Explorer

This page intentionally left blank.

GLOSSARY of TERMS for the Flight Operations Segment

activity	A specified amount of scheduled work that has a defined start date, takes a specific amount of time to complete, and comprises definable tasks.
analysis	Technical or mathematical evaluation based on calculation, interpolation, or other analytical methods. Analysis involves the processing of accumulated data obtained from other verification methods.
attitude data	Data that represent spacecraft orientation and onboard pointing information. Attitude data includes: o Attitude sensor data used to determine the pointing of the spacecraft axes, calibration and alignment data, Euler angles or quaternions, rates and biases, and associated parameters. o Attitude generated onboard in quaternion or Euler angle form. o Refined and routine production data related to the accuracy or knowledge of the attitude.
availability	A measure of the degree to which an item is in an operable and committable state at the start of a "mission" (a requirement to perform its function) when the "mission" is called for an unknown (random) time. (Mathematically, operational availability is defined as the mean time between failures divided by the sum of the mean time between failures and the mean down time [before restoration of function].

availability (inherent) (A_i)

The probability that, when under stated conditions in an ideal support environment without consideration for preventive action, a system will operate satisfactorily at any time. The "ideal support environment" referred to, exists when the stipulated tools, parts, skilled work force manuals, support equipment and other support items required are available. Inherent availability excludes whatever ready time, preventive maintenance downtime, supply downtime and administrative downtime may require. A_i can be expressed by the following formula:

A_i = MTBF/ (MTBF + MTTR)

Where: MTBF = Mean Time Between Failures
MTTR = Mean Time To Repair

availability (operational) (A_o)

The probability that a system or equipment, when used under stated conditions in an actual operational environment, will operate satisfactorily when called upon. A_o can be expressed by the following formula:

A_o = MTBM / (MTBM + MDT + ST)

Where: MTBM = Mean Time Between Maintenance (either corrective or preventive)
MDT = Mean Maintenance Down Time where corrective, preventive administrative and logistics actions are all considered.
ST = Standby Time (or switch over time)

A schedule of activities for a target week corresponding to normal instrument operations constructed by integrating long term plans (i.e., LTSP, LTIP, and long term spacecraft operations plan).

build

An assemblage of threads to produce a gradual buildup of system capabilities.

calibration

The collection of data required to perform calibration of the instrument science data, instrument engineering data, and the spacecraft engineering data. It includes pre-flight calibration measurements, in-flight calibrator measurements, calibration equation coefficients derived from calibration software routines, and ground truth data that are to be used in the data calibration processing routine.

command	Instruction for action to be carried out by a space-based instrument or spacecraft.
command and data handling (C&DH)	The spacecraft command and data handling subsystem which conveys commands to the spacecraft and research instruments, collects and formats spacecraft and instrument data, generates time and frequency references for subsystems and instruments, and collects and distributes ancillary data.
command group	A logical set of one or more commands which are not stored onboard the spacecraft and instruments for delayed execution, but are executed immediately upon reaching their destination on board. For the U.S. spacecraft, from the perspective of the EOS Operations Center (EOC), a preplanned command group is preprocessed by, and stored at, the EOC in preparation for later uplink. A real-time command group is unplanned in the sense that it is not preprocessed and stored by the EOC.
detailed activity schedules	The schedule for a spacecraft and instruments which covers up to a10 day period and is generated/updated daily based on the instrument activity listing for each of the instruments on the respective spacecraft. For a spacecraft and instrument schedule the spacecraft subsystem activity specifications needed for routine spacecraft maintenance and/or for supporting instruments activities are incorporated in the detailed activity schedule.
direct broadcast	Continuous down-link transmission of selected real-time data over a broad area (non-specific users).

EOS Data and Operations System (EDOS) production data set	Data sets generated by EDOS using raw instrument or spacecraft packets with space-to-ground transmission artifacts removed, in time order, with duplicate data removed, and with quality/ accounting (Q/A) metadata appended. Time span or number of packets encompassed in a single data set are specified by the recipient of the data. These data sets are equivalent to Level 0 data formatted with Q/A metadata. For EOS, the data sets are composed of: instrument science packets, instrument engineering packets, spacecraft housekeeping packets, or onboard ancillary packets with quality and accounting information from each individual packet and the data set itself and with essential formatting information for unambiguous identification and subsequent processing.
housekeeping data	The subset of engineering data required for mission and science operations. These include health and safety, ephemeris, and other required environmental parameters.
instrument	o A hardware system that collects scientific or operational data. o Hardware-integrated collection of one or more sensors contributing data of one type to an investigation. o An integrated collection of hardware containing one or more sensors and associated controls designed to produce data on/in an observational environment.
instrument activity deviation list	An instrument's activity deviations from an existinginstrument activity list, used by the EOC for developing the detailed activity schedule.
instrument activity list	An instrument's list of activities that nominally covers seven days, used by the EOC for developing the detailed activity schedule.
instrument engineering data	subset of telemetered engineering data required for performing instrument operations and science processing
instrument microprocessor memory loads	Storage of data into the contents of the memory of an instrument’s microprocessor, if applicable. These loads could include microprocessor-stored tables, microprocessor-stored commands, or updates to microprocessor software.

instrument resource deviation list	An instrument's anticipated resource deviations from anexisting resource profile, used by the EOC for establishing TDRSS contact times and building the preliminary resource schedule.
instrument resource profile	Anticipated resource needs for an instrument over a targetweek, used by the EOC for establishing TDRSS contact times and building the preliminary resource schedule.
instrument science data	Data produced by the science sensor(s) of an instrument, usually constituting the mission of that instrument.
long-term instrument plan (LTIP)	The plan generated by the instrument representative to the spacecraft's IWG with instrument-specific information to complement the LTSP. It is generated or updated approximately every six months and covers a period of up to approximately 5 years.
long-term science plan (LTSP)	The plan generated by the spacecraft's IWG containing guidelines, policy, and priorities for its spacecraft and instruments. The LTSP is generated or updated approximately every six months and covers a period of up to approximately five years.
long term spacecraft operations plan	Outlines anticipated spacecraft subsystem operations and maintenance, along with forecasted orbit maneuvers from the Flight Dynamics Facility, spanning a period of several months.
mean time between failure (MTBF)	The reliability result of the reciprocal of a failure rate that predicts the average number of hours that an item, assembly or piece part will operate within specific design parameters. (MTBF=1/(l) failure rate; (l) failure rate = # of failures/ operating time.
mean down time (MDT)	Sum of the mean time to repair MTTR plus the average logistic delay times.
mean time between maintenance (MTBM)	The mean time between preventive maintenance (MTBPM) and mean time between corrective maintenance (MTBCM) of the ECS equipment. Each will contribute to the calculation of the MTBM and follow the relationship: 1/MTBM = 1/MTBPM + 1/MTBCM
mean time to repair (MTTR)	The mean time required to perform corrective maintenance to restore a system/equipment to operate within design parameters.

object	Identifiable encapsulated entities providing one or more services that clients can request. Objects are created and destroyed as a result of object requests. Objects are identified by client via unique reference.
orbit data	Data that represent spacecraft locations. Orbit (or ephemeris) data include: Geodetic latitude, longitude and height above an adopted reference ellipsoid (or distance from the center of mass of the Earth); a corresponding statement about the accuracy of the position and the corresponding time of the position (including the time system); some accuracy requirements may be hundreds of meters while other may be a few centimeters.
playback data	Data that have been stored on-board the spacecraft for delayed transmission to the ground.
preliminary resource schedule	An initial integrated spacecraft schedule, derived from instrument and subsystem resource needs, that includes the network control center TDRSS contact times and nominally spans seven days.
preplanned stored command	A command issued to an instrument or subsystem to be executed at some later time. These commands will be collected and forwarded during an available uplink prior to execution.
principal investigator (PI)	An individual who is contracted to conduct a specific scientific investigation. (An instrument PI is the person designated by the EOS Program as ultimately responsible for the delivery and performance of standard products derived from an EOS instrument investigation.)
prototype	Prototypes are focused developments of some aspect of the system which may advance evolutionary change. Prototypes may be developed without anticipation of the resulting software being directly included in a formal release. Prototypes are developed on a faster time scale than the incremental and formal development track.

raw data	Data in their original packets, as received from the spacecraft and instruments, unprocessed by EDOS. o Level 0 – Raw instrument data at original resolution, time ordered, with duplicate packets removed. o Level 1A – Level 0 data, which may have been reformatted or transformed reversibly, located to a coordinate system, and packaged with needed ancillary and engineering data. o Level 1B – Radiometrically corrected and calibrated data in physical units at full instrument resolution as acquired. o Level 2 – Retrieved environmental variables (e.g., ocean wave height, soil moisture, ice concentration) at the same location and similar resolution as the Level 1 source data. o Level 3 – Data or retrieved environmental variables that have have been spatially and/or temporally resampled (i.e., derived from Level 1 or Level 2 data products). Such resampling may include averaging and compositing. o Level 4 – Model output and/or variables derived from lower level data which are not directly measured by the instruments. For example, new variables based upon a time series of Level 2 or Level 3 data.
real-time data	Data that are acquired and transmitted immediately to the ground (as opposed to playback data). Delay is limited to the actual time required to transmit the data.
reconfiguration	A change in operational hardware, software, data bases or procedures brought about by a change in a system's objectives.
SCC-stored commands and tables	Commands and tables which are stored in the memory of the central onboard computer on the spacecraft. The execution of these commands or the result of loading these operational tables occurs sometime following their storage. The term "core-stored" applies only to the location where the items are stored on the spacecraft and instruments; core-stored commands or tables could be associated with the spacecraft or any of the instruments.
scenario	A description of the operation of the system in user's terminology including a description of the output response for a given set of input stimuli. Scenarios are used to define operations concepts.